A SPREADSHEET MODEL FOR USING WEB SERVICES AND CREATING DATA-DRIVEN APPLICATIONS

KERRY SHIH-PING CHANG

April 2016

CMU-HCII-16-100

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

THESIS COMMITTEE

Brad A. Myers (Chair), Carnegie Mellon University
Niki Kittur, Carnegie Mellon University
John Zimmerman, Carnegie Mellon University
Margaret M. Burnett, Oregon State University

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Copyright © 2016 Kerry Shih-Ping Chang. All rights reserved.

This research was funded in part by the NSF under grants IIS-1116724 and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

Keywords: spreadsheets, web services, data-driven applications, web applications, streaming data, hierarchical data, end-user programming, end-user software engineering (EUSE), Natural Programming

ABSTRACT

Web services have made many kinds of data and computing services available. However, to use web services often requires significant programming efforts and thus limits the people who can take advantage of them to only a small group of skilled programmers. In this dissertation, I will present a tool called *Gneiss* that extends the spreadsheet model to support four challenging aspects of using web services: programming two-way data communications with web services, creating interactive GUI applications that use web data sources, using hierarchical data, and using live streaming data. Gneiss contributes innovations in spreadsheet languages, spreadsheet user interfaces and interaction techniques to allow programming tasks that currently require writing complex, lengthy code to instead be done using familiar spreadsheet mechanisms. Spreadsheets are arguably the most successful and popular data tools among people of all programming levels. This work advances the use of spreadsheets to new domains and could benefit a wide range of users from professional programmers to end-user programmers.

ACKNOWLEDGEMENTS

I am sincerely grateful to my advisor, Brad Myers, for his invaluable advice. This research would not be possible without his guidance and support. Brad is one of the sharpest and most knowledgeable researchers that I have known. He is also a caring and patient teacher. Brad taught me how to do research and showed me how fun and exciting it could be. My six years in CMU have been a wonderful journey of learning and growing because of him, and I cannot thank him enough for that. I will miss our time working together.

I thank my committee members John Zimmerman, Margaret Burnett and Niki Kittur for their time and feedback. The new perspectives they brought in helped me tremendously in shaping my proposed work and in structuring and motivating this thesis. I also thank my undergraduate advisor Hao-Hua Chu for introducing me into human-computer interaction and giving me the first sweet taste of doing research. I would not pursue this degree without his encouragements.

My fellow HCII cohorts Ruogu Kang and Yanjin Long have been there with me through all the ups and downs in graduate school. No words can describe how thankful I am to them, and I will forever cherish our friendship. A big thank you to my friends Ross Grieshaber, Da-Cheng Juan and Gary Pan. My time in Pittsburgh would not be as memorable without you. Mostly, I would like to thank my family – my mother Kuo-Mei, my sister Hui-Ping and my brother Sung-Ping – for their continuous love and support.

TABLE OF CONTENTS

Abstract iii
Acknowledgements iv
Table Of Contents v
List Of Figures And Tables ix

1. Introduction 1
1.1 The Popularity Of Web Data Services, And The Challenges 1
1.2 An Approach: Spreadsheet Programming 2
1.3 Gneiss: A Spreadsheet Tool For Using Online Data 2
1.3.1 A Model For Using Web Services 2
1.3.2 A Model For Creating Interactive Web Data Applications 3
1.3.3 A Model For Using Hierarchical Data 4
1.3.4 A Model For Using Streaming Data 5
1.4 Contributions 5
1.5 Thesis Statement 6
1.6 Outline 7

2. Related Work 8
2.1 Motivating Research 8
2.1.1 Why Spreadsheet Programming? 8
2.1.2 End-User Programmers 10
2.2 Mashup Tools 13
2.3 Tools For Creating Data-Driven Applications 14
2.3.1 Spreadsheet Tools For Making Graphical User Interfaces 17
2.3.2 Model-View Controller Libraries 18
2.3.3 End-User Web Programming And Visualization Tools 19
2.4 Tools For Using Hierarchical Data 19
2.5 Tools For Using Streaming Data 21
2.6 Conclusions 23

3. Using Web Data Services 24
3.1 Motivation, Challenges And Contributions 24
3.2 Interface Overview 25
3.3 Usage Scenario 27
3.4 Key Features For Using Web Services 29
3.4.1 Selecting And Extracting Web Service Data 29
3.4.2 Sending Spreadsheet Data To Web Services 30
3.4.3 Sorting And Filtering Web Service Data 32
3.4.4 "Autofilling" Cells With Web Service Data 33
3.4.5 Using Structured Data 34
3.4.6 Refreshing Web Service Data 35

3.4.7 Parallel-Running Programs, Error Handling And Maintenance....36
3.5 Demonstrative Examples 37
3.5.1 City Trip Planner With A Map....37
3.5.2 Customized Book List....38
3.6 Limitations And Discussion 39
3.6.1 Spreadsheet Usability....39
3.6.2 Supporting More Types Of Web Services....39
3.7 Spinel: Adding Data Sources To An Application As Plugins 40
3.8 Conclusions 44
4. Programing Data-Driven Web Applications 45
4.1 Motivation, Challenges And Contributions 45
4.2 Usage Scenario 47
4.3 Key Features For Creating Interactive Web Applications 50
4.3.1 Creating A Dynamic UI That Shows Spreadsheet Data....50
4.3.2 Modifying Spreadsheet Data From Web Applications....53
4.3.3 Making Applications Be Interactive....54
4.4 Demonstrative Examples 57
4.4.1 Photo Slideshow....57
4.4.2 Animations....59
4.4.3 Posting Data....59
4.5 Limitations And Discussion 59
4.6 Conclusions 61
5. Using Structured Hierarchical Data....62
5.1 Motivation, Challenges And Contributions 62
5.2 Usage Scenario 65
5.3 Key Features For Using Hierarchical Data 69
5.3.1 Displaying Hierarchical Data In Spreadsheets....69
5.3.2 Manipulating Hierarchical Data....73
5.4 Demonstrative Example 79
5.4.1 Paper Geography Visualization....79
5.5 User Study 81
5.5.1 Study Design....81
5.5.2 Participants....81
5.5.3 Data....82
5.5.4 Tasks....82
5.5.5 Procedure....83
5.5.6 Results....83
5.6 Discussion 85
5.6.1 Learning And Using Gneiss....85
5.6.2 Strengths Of Having Hierarchies In Spreadsheets....86
5.6.3 Strengths Of Having A Visual Tool To Work With Data....87
5.6.4 Lack Of Computational Thinking....87

5.6.5 POSSIBLE FORMS OF OUTPUTS 88
5.7 Limitations 88
5.8 Conclusions 89
6. Using Streaming Data 90
6.1 Motivation, Challenges And Contributions 90
6.2 Usage Scenario 91
6.3 Key Features For Using Streaming Data 93
6.3.1 Spreadsheet Cell Metadata 93
6.3.2 Controlling Streaming Timing 95
6.3.3 Stream Data From Web Input Elements 96
6.4 Demonstrative Examples 96
6.4.1 Real-Time Weather Alerts Based On Locations 97
6.4.2 Online Exam Page 97
6.5 Limitations And Discussion 98
6.6 Conclusions 100
7. Implementation 101
7.1 Client-Server Architecture 101
7.1.1 Live Frontend Editor Implemented With Constraints 101
7.1.2 Interacting With Web Services At The Server 104
7.1.3 Saving And Reopening A Spreadsheet 104
7.1.4 Exporting A Web Application 105
7.2 Internal JSON Data Models 106
7.3 Conclusions 110
8. Future Work 111
8.1 Gneiss Editor Features 111
8.1.1 General Usability Features 111
8.1.2 Spreadsheet Functions 111
8.1.3 Spreadsheet Sorting And Filtering 112
8.1.4 Spreadsheet Autofilling 112
8.1.5 Web Interface Builder Features 113
8.1.6 Accepting More Types Of Data Sources 113
8.2 Server-Side Extensions 113
8.2.1 Moving Client-Side Computations To The Server 113
8.2.2 Sharing Vs. Independent Web Application Data 114
8.3 Improving The Programming Of Interactive Behaviors 114
8.4 Spreadsheets As Database Consoles 115
8.5 Data-Driven Mobile Applications 116
8.6 Debugging 116
8.7 Incorporating Gneiss Into Current Programming Practices 117
8.8 Understanding End-User Programmers In The Real World Through Gneiss 118
8.9 Conclusions 119

9. Conclusions 120

Appendix A. A List of supported spreadsheet functions in Gneiss 132
Appendix B. A List of supported web GUI elements in Gneiss 134
Appendix C. Post-Study Survey for the user study 136

LIST OF FIGURES

Figure 1.1. Gneiss's user interface consists a browser-like source pane (1), a spreadsheet editor (2), and a web interface builder (3). **3**

Figure 1.2. Gneiss visualizes hierarchies in data using nested spreadsheet cells (1), and lets users restructure the data by any fields by drag-and-dropping a column to a different location (2 and 3). **4**

Figure 3.1. A screenshot of Gneiss's interface. **26**

Figure 3.2. A screenshot of Gneiss showing the spreadsheet program created in the usage scenario. At the left is a source pane that shows the raw data returned from a web service. (1) is the URL textbox where the user enters the address of a web API. Note that the value of the cell A1 has been used as the search term. (2) is the returned data. (3) is the spreadsheet interface where the user can store desired fields extracted from the raw web service data and do manipulations. (4) If the extracted data have structure, they are shown in nested tables. The final results, the driving time from Alice's school to the restaurant, are shown in column F at (5). **27**

Figure 3.3. Pressing the arrow icon at the top of each column brings up a dialog box that lets the user apply sorting and filtering to that column. Data extracted from the same source (in neighboring columns) are sorted and filtered together and are highlighted with a purple background. The column from which the dialog box is opened is highlighted with a purple border. **28**

Figure 3.4. A screenshot of the demonstrative example in section 3.5.1 **38**

Figure 3.5. Spinel's architecture and modules **41**

Figure 3.6. A UML sequence diagram showing a use of the Spinel Library. Time goes down from the top. **42**

Figure 3.7. The main API editing page in the Spinel web-based plug-in configuration tool for users. (1) An API is divided into a base URL and a set of parameters, which are filled in by the tool if the user provided an example API request on a previous page (not shown). (2) The user selects a parameter to be an input field and gives it an appropriate name using the drop-down menu that is generated based on the developer's instructions. (3) The user can click on the "Test this API" button to see the return data (4, with the gray background). The user can select the desired fields as the final output of this API call using the checkboxes. The tool shows all selected field names and visualizes the output model that will be sent to the application in a tree at the far right, to help the user confirm that her selections are correct. **43**

Figure 4.1. A screenshot of Gneiss showing the web application created in this chapter's usage scenario. (1) is the source pane where the user can load a web API in the URL box and extract the desired fields from the return data to the spreadsheet editor through drag-and-drop. (2) is the spreadsheet editor that stores and manipulates the data to be used in the web application. (3) is the web interface builder where the user can create a web application by dragging-and-dropping GUI elements from the toolbar on the right (4) to the output page. The user can select a GUI element in the output page (the selected element is highlighted with a dark blue border, which currently is the textbox at the top of the page) and view its properties in (5). Property values are cells that can contain formulas and can be referenced by other cells. **47**

Figure 4.2. Sorting and filtering rules can be computed from GUI element properties in a web application using a spreadsheet formula. **49**

Figure 4.3. The user can display hierarchical data in the spreadsheet (left) as nested lists in the web application (right). Here, the user creates a first-level list to show restaurant names (column A), and within each restaurant in the first-level list the user drags a second list that displays the categories of the restaurant (column B)**52**

Figure 4.4. A screenshot of making a slideshow interface with photos from Flickr. At the left, column P stores a list of photo URLs, Q1 stores the current slideshow index (3), and Q2 stores the button that the user last clicked on ("Next"). At the right is the slideshow application. The thumbnails show the photos in column P. The large image is the photo at row 3 (the index value stored in Q1) and the 3rd thumbnail is highlighted**57**

Figure 5.1. (1) A JSON object describing the Cleveland Cavaliers. The coaches and players are shown in two arrays. (2) Converting this JSON object into a "long table" spreadsheet format where the hierarchies are expended vertically creates a lot of repeated values, whereas (3) converting it into a "wide table" spreadsheet format where the hierarchies are expended horizontally creates a lot of columns with empty cells**63**

Figure 5.2. The JSON object in Figure 5.1 at 1 shown in Gneiss. Gneiss visualizes the hierarchies of data using nested cells with nested row labels to allow users to select data by its structure. In this JSON object, the team coaches and players are in two different arrays. Therefore, Gneiss uses a thin grey line between them (column B and C) to show that they are not hierarchically connected (same as the player's positions and NBA champion years in column E and F)**64**

Figure 5.3. Two example JSON objects used in the usage scenario. The left JSON object describes a conference paper. The right JSON object describes a paper session in a conference, with each paper listed in order, using its ID**65**

Figure 5.4. Six tables showing how the user can compute summaries of a JSON file of conference papers. (1) The data initially is indexed by paper titles. Each paper has multiple authors. Each author has an institution. The user regroups the data by institutions by dragging the institutions to the beginning of the table (2), right-clicks column A and chooses "Group Column B-C by Column A" (3) to merge rows that have the same value in column A (4). Column B now has many repeated values because a paper can have many authors from the same institution. (5) The user groups the data again by column B to merge repetitive paper titles within an institution. Lastly, (6) the user inserts a new column at column B, enters =COUNT(C1) in B1 to get the paper count for the first institution, and autofills the value for the rest of the institutions. She sorts the data by column B to bring the institution that has the most papers to the top**66**

Figure 5.5. A screenshot showing the result spreadsheet of the second task in the usage scenario. The left pane is where the user loads and views a JSON file (through a URL bar for online data or a "Choose file" button for local data). The spreadsheet is a custom conference schedule created by the user. The spreadsheet shows for each time slot, the session that has the most number of award papers plus papers that have "crowdsourcing" in its keywords. The data is grouped by day (column A) and time (column B). Column C shows the title of the session that the user should go to. The number of award plus crowdsourcing papers of that session is in column D, with the details of the papers in the rest of the columns**67**

Figure 5.6. The user joins two tables by selecting a common column in each table and clicking the "Join" option from the right-click menu. The common columns (here, columns D and E) are combined into one column after the tables are joined. **68**
Figure 5.7. The user can insert a new blank column that has the same structure as a selected column. **68**
Figure 5.8. An old design of Gneiss's nested table. **69**
Figure 5.9. The new design of Gneiss's nested table, showing similar data as in Figure 8. ..... **70**
Figure 5.10. (1) shows countries funded by the World Bank. Each country has a list of major sectors and the percentage of funding each sector received. The user can sort and filter the data by any column using a dialog box (2). When sorting and filtering on a nested column, the operation is executed within each nested table, marked in the spreadsheet with blue borders (3). The system highlights the affected columns using a purple background when the dialog box is open. **77**
Figure 5.11. An example of joining two hierarchical objects. The two trees are joined by the yellow nodes. (Left) The blue tree has the yellow nodes in a deeper hierarchy level than the red tree. (Right) The join operation preserves the blue tree's structure and connects the red tree to the blue tree through the matched yellow nodes (shown as the yellow nodes that have two rings) **78**
Figure 5.12. A screenshot of this demonstrative example. In the spreadsheet, the user groups the CHI'15 papers by their continents, countries and institutions. The user then visualizes the spreadsheet data using a treemap in the right web interface builder. Through the treemap, the user can view the distribution of data in each hierarchical level. For example, she can tell from the treemap that in Asia, Japan, Taiwan and Singapore are the countries that have the most papers. **80**
Figure 5.13. The average task completion time for the Gneiss, Excel and programming groups. Shorter bars are better. **84**
Figure 6.1. The user is creating a real-time web application (the right pane) that streams Yahoo's stock prices (columns A and B in the spreadsheet) every 10 minutes. To let Gneiss periodically pull data from a web service, the user selects the "Stream this source" checkbox (1) in the source pane. The system shows the source pane data's last retrieved time in a blue label. Gneiss's spreadsheet allows users to set custom streaming frequency and pausing rules, and to sort and filter the data by fetched time (2). In this screenshot, the user uses a line chart (3) to see the ups and downs of today's prices. Full description of this example screenshot is in section 6.2. ..... **91**
Figure 6.2. When entering a time string to filter the data, Gneiss allows flexible formats, parses the user's input and shows the system interpreted time in grey text below the textbox. **94**
Figure 6.3. The user can also stream data from web input elements to a spreadsheet column, by having the streaming source to be a GUI element property. **96**
Figure 7.1. A hierarchical table visualized in nested cells in Gneiss. **106**

LIST OF TABLES

Table 3.1. Computed rules for sorting and filtering supported in Gneiss. **.....32**

CHAPTER 1 INTRODUCTION

We live in a world of digital data where all kinds of public and personal data are stored in cloud databases and can be accessed through the Internet. While some data are presented in the form of web pages, many data sources have also provided *web services* that let people use their data programmatically. Web services offer a more efficient and reliable way to collect online data than scraping web pages. Some web services not only provide data but also computing services that can analyze user data using powerful cloud computing backend. As a result, web services have become the major way for people to create custom applications or to do custom data analyses using the cloud or data in the cloud.

1.1 THE POPULARITY OF WEB DATA SERVICES, AND THE CHANLLENGES

Today, a person can find almost any type of data provided by at least one web service. For example, Google provides web services to a variety of data including public data such as web search results, videos, places, real-time data such as finance and traffic information, and even personal data such as calendars; Twitter and Facebook provide web services to social network data; Zillow provides web services to real estate data; Rotten Tomatoes and Last.fm provide web services to movie and music data; ESPN provides web services to sport data; and Nike Plus provides web services to personal health data collected by wearable devices. Some web services provide the ability to transform or analyze the received user data. For example, GeoNames' web service accepts geo coordinates and turns them into country codes and postal codes. Amazon provides Machine Learning API that lets users run machine learning algorithms on their data. As of March 2016, ProgrammableWeb.com lists over 14,543 web services in over 400 categories, and over 6,250 custom applications that use those web services.

The amount of web data services continuously increases. However, currently only a small group of people can freely take advantage of those services because using them often requires significant programming efforts. A person needs to write code to send API calls over the Internet, parse the return data and manipulate it into the correct form to create desired analyses and presentations. For professional programmers, writing the necessary code to do these things could be tedious, error-prone and requires significant effort and learning [96]. For many end-user programmers (EUPs) [50] who have the need to make custom use of web data, the programming barriers are often too difficult to overcome, so they would either give up or try to find professional programmers to help, which can be costly and time-consuming [57].

1.2 AN APPROACH: SPREADSHEET PROGRAMMING

My dissertation focuses on extending spreadsheet programming to facilitate the use of online data and data services. The reason for choosing spreadsheets is obvious: spreadsheets are the most successful and pervasive data tool. They are already popular among users of all programming levels [69]. The spreadsheet's table interface, its language syntax, functions and its live programming have been shown to be friendly to learn and use especially for end-user programmers. Features in the conventional spreadsheet model have been used by millions of people throughout the years and many usability problems have been removed. Leveraging this familiar model and extending it to support new programming activities could mean having a higher chance of success and making a greater impact through benefiting the spreadsheet's large user base, ranging from professional programmers and data analysts to end-user programmers.

1.3 GNEISS: A SPREADSHEET TOOL FOR USING ONLINE DATA

My dissertation presents a new spreadsheet model that supports four challenging aspects of using web services: *exchanging data with web services, creating interactive applications that use web services, using structural hierarchical data* such as JSON data, and *using live streaming data*. This new spreadsheet model is realized in an interactive programming environment that I created called *Gneiss*[1] (Figure 1.1). Like conventional spreadsheets, Gneiss is a live programming tool [81] where new values are distributed throughout the program and reflected in the output as soon as the user makes an edit. It uses a "programming-with-example" [64] style as it allows the user to develop programs using visible example values from real data sources. Gneiss introduces the following innovations:

1.3.1 A MODEL FOR USING WEB SERVICES

First, my dissertation extends spreadsheets to support exchanging data with web services. To use web service data otherwise requires a programmer to write asynchronous network callbacks to send API requests and wait for the returned data. The code for the callbacks is often lengthy and complicated structurally [70] as the return of a request could trigger other requests to be sent or cause some data to be recomputed. The programmer also needs to write additional code to parse the returned document and extract the desired data. Some conventional spreadsheets provide functions to fetch data from the web given an URL. However, the returned document is put into a single cell as plain text and becomes almost unreadable and unusable. Parsing the returned data requires writing non-spreadsheet code.

[1] Gneiss (pronounced the same as "nice") is a kind of rock. Here it stands for "Gathering Novel End-user Internet Services using Spreadsheets". Publications, demo videos and the source code of Gneiss can be found at http://www.cs.cmu.edu/~shihpinc/gneiss.html.

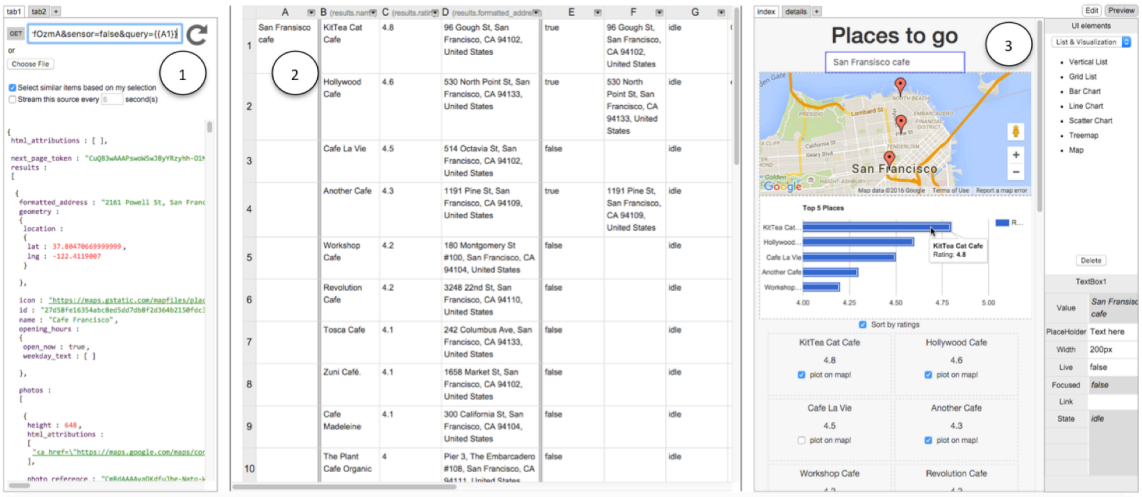


Figure 1.1. Gneiss's user interface consists a browser-like source pane (1), a spreadsheet editor (2), and a web interface builder (3).

Gneiss provides a model for utilizing data from arbitrary REST web services that allows users to construct *two-way data flows* between multiple web services and a spreadsheet without writing conventional code. It integrates the spreadsheet environment with a "source pane" (Figure 1.1 at 1) lets users execute web API requests and view the return data. Spreadsheet languages and interaction techniques replace event callbacks and query languages to send data to and retrieve and extract data from web services. Gneiss further leverages the spreadsheet's constraint evaluation to handle different states of asynchronous network requests, enabling independent web service calls to run in parallel. This part of the work is described in Chapter 3.

1.3.2 A MODEL FOR CREATING INTERACTIVE WEB DATA PPLICATIONS

One of the common uses of web services is for creating custom web pages or *web applications* that use online data. ProgrammableWeb.com lists over 6000 such applications. Some of them provide new search features such as aggregating data from multiple sources, and some of them provide new ways to understand data such as sharing new data analysis results and visualizations. Those applications usually have many interactive features to let users perform custom operations on the backend data, such as setting their own sorting and filtering rules, viewing the data in visualizations, and have the content dynamically generated based on the user's actions. Creating such applications would require a programmer to write a lot of JavaScript code. Current spreadsheets can let users create static charts and graphs, but they do not support programming web applications that use web data sources and have custom interactivity.

Gneiss unifies the properties of the elements of web applications with the spreadsheet model, so the user can create web applications that dynamically use

and present spreadsheet data using the familiar spreadsheet languages. It extends the spreadsheet language syntax to let properties of a GUI element created in a web interface builder (Figure 1.1 at 3) be referenced and used in spreadsheet formulas in the same way as spreadsheet cells, allowing the user to program two-way data bindings between the web application and the spreadsheet. Instead of using event-based ("push-based") programming, it contributes a way to program interactivity in web applications using the spreadsheet's equation-based ("pull-based") evaluation model. This allows users to program data-related interactive behaviors in a web application such as retrieving new data from sources, storing user inputs, and interactively sorting, filtering and visualizing data, all using the familiar spreadsheet mechanisms. This part of work is described in Chapter 4.

1.3.3 A MODEL FOR USING HIERARCHICAL DATA

After users acquire data from web services, they must have ways to properly manipulate and analyze the data, or the data will not be useful to them. The majority of modern web services return data in hierarchical formats, such as JSON and XML[2]. However, current spreadsheets do not support hierarchical data formats. The only way to use a JSON or XML document in spreadsheets is to convert it into a flat table. But this method would inevitably create much repetitive data in the table because spreadsheets have no notion of hierarchies. Flat tables also make it impossible to select and manipulate data using the hierarchical structure.

My dissertation extends the spreadsheet model to support using hierarchical data, defining how hierarchical objects can be shown and manipulated in spreadsheets. Instead of removing the hierarchies in data to flatten it to a table, this new spreadsheet model visualizes hierarchies in data and uses them to facilitate data manipulation and exploration. Under this model, the user can use interaction techniques to reshape (Figure 1.2), regroup or join hierarchical objects in a spreadsheet. This model also extends spreadsheet languages, sorting and filtering to support selecting and manipulating data by its hierarchies, allowing the user to calculate summaries of data using spreadsheet formulas without the need of pivot tables. This part of work is described in Chapter 5.

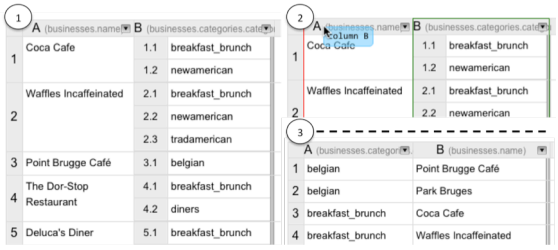


Figure 1.2. Gneiss visualizes hierarchies in data using nested spreadsheet cells (1), and lets users restructure the data by any fields by drag-and-dropping a column to a different location (2 and 3).

[2] See http://www.programmableweb.com/apis/directory.

1.3.4 A MODEL FOR USING STREAMING DATA

Streaming data is another type of online data that has become increasing popular. Data like weather, traffic, finance information, and social network feeds change live and often need to be collected and analyzed in real-time. While there are many programming libraries (e.g., [3,32,97]) that help developers in getting streaming data from data sources and analyzing them, using those libraries requires significant programming effort and expertise. There are plugins for conventional spreadsheets that let users stream data from specific data sources to a spreadsheet without writing code (e.g., [83,98]). But adding new data sources to use with those plugins requires writing non-spreadsheet code.

Gneiss's new spreadsheet model allows users to stream data from arbitrary REST web services to a spreadsheet. It shows streaming data using live columns whose length grows as new values come in, enabling users to easily program live analyses using spreadsheet formulas. As streaming data is essentially time-series data, Gneiss integrates the concept of "time" into the spreadsheets. Gneiss introduces the concept of "spreadsheet cell metadata" that allows each spreadsheet cell to have metadata that describe attributes other than its value, such as the value's fetch time. Cell metadata can be used to manipulate spreadsheet data, enabling the user to manipulate streaming using temporal information such as to compute the average of only the data which arrived within a certain time period. This part of the work is described in Chapter 6.

1.4 CONTRIBUTIONS

This dissertation contributes a new spreadsheet model and a tool that implements this model to facilitate the use of online data. Specifically, this dissertation contributes:

- A spreadsheet language to send data in arbitrary spreadsheet cells to arbitrary REST web services (Chapter 3).
- An interaction technique to extract or stream arbitrary fields in structured data returned from web services to spreadsheet columns (Chapters 3 and 6).
- An extension to spreadsheets to support sorting and filtering dynamic values returned from web services (Chapter 3).
- An extension to spreadsheet's autofilling gesture for sending similar web API requests by example (Chapter 3).
- A spreadsheet language to program data bindings between web GUI elements and a spreadsheet (Chapter 4).
- As a result of the above contributions, my dissertation contributes a spreadsheet environment that supports constructing two-way data flows among REST web data services, web applications and spreadsheets.

- An extension to spreadsheets to allow the sorting and filtering rules to be computed from web GUI controls and spreadsheet cells (Chapter 4).
- Dynamic web GUI elements that can be shown and hidden at run time based on the data in the spreadsheet (Chapter 4).
- An extension to spreadsheet's autofilling gesture for referencing dynamic web GUI elements in spreadsheet formulas (Chapter 4).
- A framework for programing interactive behaviors in web applications using spreadsheet's equation-based syntax (Chapter 4).
- A method to dynamically visualize hierarchical data in spreadsheets based on the hierarchical relationship among columns (Chapter 5).
- Interaction techniques to reshape, regroup and join hierarchical data in a spreadsheet (Chapter 5).
- A method to connect multiple hierarchical objects by common fields without flattening the objects (Chapter 5).
- A spreadsheet language for using hierarchical data in spreadsheet formulas, allowing the user to calculate summaries of data without using pivot tables (Chapter 5).
- Evidence from a lab study which showed that Gneiss helped spreadsheet users explore and analyze hierarchical datasets significantly faster than Excel and than programmers writing JavaScript or Python code (Chapter 5).
- A method for visualizing streaming data in spreadsheets (Chapter 6).
- A design for spreadsheet cell metadata that records attributes of a cell's value, such as a cell's provenance and fetched time (Chapter 6).
- New spreadsheet functions that can access a cell's metadata and select cells using their metadata (Chapter 6)
- An extension to spreadsheet's sorting and filtering mechanisms to support manipulating spreadsheet cells using their metadata, such as to sort and filter a column using temporal information (Chapter 6).

1.5 THESIS STATEMENT

My thesis statement is:

> A new spreadsheet model can enable the programing of interactive web applications and data analyses that use hierarchical data and streaming data from web services.

To evaluate this statement, I used Gneiss to create a series of example applications to demonstrate the ability and range of this spreadsheet model. Those examples are described in Sections 3.3, 3.5, 4.2, 4.4, 5.2, 5.4, 6.2 and 6.4.

I also conducted a lab study where I recruited intermediate spreadsheet users who are not professional programmers to use Gneiss or Excel to analyze hierarchical

JSON documents. In the same study I also recruited professional programmers to write JavaScript or Python to analyze the same data. On average, Gneiss helped spreadsheet users complete study tasks almost two times faster than Excel, and they even outperformed professional programmers in most tasks. The study is described in Section 5.5.

1.6 OUTLINE

The rest of the dissertation is organized as follows: Chapter 2 is related work. Chapter 3 describes how Gneiss supports using web services. Chapter 4 describes how Gneiss supports creating data-driven applications. Chapter 5 describes how Gneiss supports using hierarchical data. Chapter 6 describes how Gneiss supports using streaming data. Chapter 7 describes the implementation and architecture of Gneiss. Chapter 8 discusses the future directions for this research. Chapter 9 concludes this dissertation by revisiting its contributions.

CHAPTER 2 RELATED WORK

This dissertation is motivated by research in spreadsheet programming and end-user programming. It is also related to professional and end-user tools for using web services, hierarchical data, streaming data and programming data-driven web applications.

2.1 MOTIVATING RESEARCH

This dissertation presents a new spreadsheet model for using online data. Upon reading this, one might ask two questions: First, why spreadsheets? Second, who will benefit from having this spreadsheet model?

2.1.1 WHY SPREADSHEET PROGRAMMING?

The success of spreadsheet programming has been a subject of studies in many publications. Here I discuss three characteristics of spreadsheets that inspire the design and creation of Gneiss. Those are the *directness* and *liveness* of spreadsheet programming, and the *modularity* of spreadsheets that may be extended to support creating templates for data-driven applications.

Much research attributed the ease of learning and use of spreadsheet programming to the *directness* of both its language [69] and its programming interface [11,42,78]. Nardi [69] observed that the formula-based syntax was straightforward to users familiar with numerical manipulation, and the spreadsheet functions provided a direct way for people to perform high-level operations without having to attend to low-level programming details, allowing users to focus on the task at hand. An example described by Nardi was to calculate the sum of a list of values: instead of having to initialize a variable and write a `for` loop to add up items in an array manually, in spreadsheets the user could do so by simply typing something like `=SUM(A1:A10)` in a spreadsheet cell. The spreadsheet interface is a visual programming environment [64]. Unlike in many professional languages such as Java or Python where the data is hidden, in spreadsheets the users can directly see the data while programming. The spreadsheet interface supports many direct manipulation techniques [78] to let people manipulate spreadsheet data without writing code, such as the autofilling (select-and-drag) gesture to fill in spreadsheet cells based on examples.

I chose to extend the spreadsheet model with the goal of bringing this directness of spreadsheet programming into other data-related programming activities, such as analyzing more kinds of data and creating data-driven applications. Often times when programming applications that use databases, the developers have to work with dynamic data – they do not have the actual data and cannot see the data when

writing code, as the data are retrieved by the users at run time. In contrast, spreadsheets allow me to create a visual environment where users can use visible example data to construct the computation logic and the look of the program. I was also able to leverage spreadsheet languages and interaction techniques to avoid writing lower-level, non-task-related code in my system. For example, in Gneiss the programmer can type the formula `=IF(Checkbox1!Checked, "Descending", "Ascending")` into the formula textbox in a sort widget to create the interactive behavior of sorting a list of data by checking a checkbox in a web page. In contrast, in JavaScript, the programmer would have to declare an event handler for the checkbox, and write functions or loops to sort the data inside the handler. Another example is extending the drag-and-drop gesture to reshape a JSON object that currently requires writing multiple (and often nested) loops to traverse through the object and change its structure.

The spreadsheet is also a *live programming environment*. It provides immediate visual feedback to the programmers after they make an edit [11,13,81]. Research has shown that live programming environments allow developers to quickly switch between editing and testing a program and help them find and fix bugs more quickly [52]. For using online data, the live nature of spreadsheets provides an opportunity for handling streaming data sources that change live. It also allows the applications created to be interactive by default. In many examples described in this dissertation, the user takes advantages of Gneiss' live programming environment not just to test or debug their programs but also to retrieve new data that they need in an interactive manner.

Finally, conventional spreadsheets have a known way to be modularized and reused– through spreadsheet templates. Spreadsheet templates have been widely used by people to share expert knowledge [68] and avoid errors [1]. In web programming, HTML and CSS templates are common ways for novice programmers to reuse the look of a web page in a WISIWIG editor. However, HTML and CSS templates can only support very limited interactivities (namely, hyperlinks and hovering) and cannot handle dynamic data from databases. Developers need to go back to a text editor to write JavaScript code to program those behaviors.

By extending spreadsheets to support programming data bindings between frontend user interfaces and backend data sources, my dissertation opens the possibility to reuse data-related interactive behaviors in the form of a spreadsheet template that can be used in a live visual programming environment. Many database applications people use daily have common interactive behaviors, such as searching the database through a textbox, sorting and filtering the data using checkboxes and sliders, and changing between a list view and a map view [17]. One can imagine such a web application becoming a template in Gneiss with a HTML/CSS template in the web interface builder at the right for the look of the application, and a spreadsheet

template in the middle that has all the computation logic of how the data should be connected to and manipulated by the web elements. Users can easily swap the data sources used in the spreadsheet to make the application their own, such as connecting to a different web service or importing a local file. This idea is inspired by prior spreadsheet tools that support creating and reusing custom GUI objects and interactive behaviors, such as Forms/3 [11] and InterState [71]. But I would argue that the spreadsheet reuse in Gneiss would be the kind that is more familiar to spreadsheet users – that is, applying a template to manipulate *their own data*.

2.1.2 END-USER PROGRAMMERS

To answer the second question, "who will benefit from this research", I review prior literature on end-user programmers. By leveraging the spreadsheet model, this research aims to provide a more intuitive way of programming to support using and publishing data on the web and thus bring the ability to do these things to end-user programmers (although professional programmers may benefit as well).

Myers et al. [61] defined end-user programmers as "people who write programs, but not as their primary job function – they write programs in support of achieving their main goal, which is something else, such as accounting, designing web pages, office work, scientific research, entertainment, etc." Ko et al. [50] defined end-user programmers as people who program to achieve personal goals, not to create software that is intended to be used by the public. Nardi [69] described end users as anyone who uses computer software. End-user programmers are people who program to achieve their computational needs, but are not interested in doing programming for a living. In contrast, professional programmers are people whose job is to write code and create computer software for other people to use. Scaffidi et al. [77] estimated that in 2012 in the USA there were over 13 million end users who would say that they "do programming" at work, while the number of professional programmers was estimated to be only 3 million.

Based on these definitions, it seems incorrect to assume that end-user programmers are "'novice' or 'naïve' users" [69]. It has more to do with a person's *intent* when she programs, instead of her programming experience. In fact, studies have shown that end-user programmers vary greatly in programming experience [50,54,67]. For example, Lawson et al.'s study [54] of about 1600 spreadsheet users in multiple organizations found almost the same amount of novice spreadsheet users and expert spreadsheet users, with most people being intermediate users. The literature describes end-user programmers as people who have strong domain knowledge, such as scientists and engineers [31], financial analysts [54] and teachers [6]. Since programming is not their profession but rather a way to help them achieve other personal or professional goals, end-user programmers often prefer to use domain-

specific languages (such as spreadsheets) to program since those languages are easier for them to learn and address their goals more directly [61,69].

My dissertation extends the spreadsheet model while keeping it as a domain-specific tool focusing on using data. Although many new concepts introduced in this dissertation will require users to learn and practice to use correctly, prior research has shown that end users could and would learn programming if the programming language matches closely the users' domain knowledge and their interests [69]. So to identify targeted end-user programmers, I argue that we look at the literature on the *needs* of end users for programming instead of their programming expertise and if they were able to learn new things. After all, spreadsheets have been shown particularly friendly to learn, as discussed in section 2.1.1.

Prior research shows that end users have the need to work with database data. For example, Chambers at al. [14] analyzed 400 spreadsheets randomly selected from a nearly 4500 end-user spreadsheet corpus. They found that 25% of the spreadsheets were exported from databases for end users to use and analyze in Excel. Scaffidi et al. [77] estimated that in 2012 there were over 55 million end users who used spreadsheets and databases at work in the USA. Another way to look for evidence of this is to look at the large amount of commercial tools that help end users construct SQL queries in a GUI and export the data as spreadsheets for further manipulation, such as Microsoft Access's Query Design view [99], SQLyog [100] and SQLeo [101]. This dissertation aims to help those spreadsheet/database users. Since more and more web databases now provide hierarchical data (for example, MongoDB, a JSON database popular among web developers, has been one of the fastest growing database systems in recent years[3]) and streaming data, extending current end-user data tools (such as spreadsheets) to support these new data types could have great value.

This dissertation also presents a visual environment to interactively query web data sources through web services and collect data in a familiar spreadsheet manner that provides many ways to do analysis and visualizations on the data. A study by Van Kleek et al. [49] found that people regularly use multiple data sources on the Internet to complete everyday tasks. For example, people would repeatedly gather information from multiple sources to validate the correctness or accuracy of the information, such as comparing product reviews on multiple shopping websites. Another use case was to reference multiple sources to help make a decision, such as Google, social network sites and dedicated reviews websites (like Yelp). The same study also found that to integrate data from multiple web services was a challenge for end-users, as the same information from different sources was often named differently or recorded in different structures. Gneiss' ability to let users use web

[3] See http://db-engines.com/en/ranking

services without writing conventional code allows users to create reusable spreadsheet programs that directly get data from multiple data sources instead of having to manually gather and compare data from multiple websites. As Gneiss provides solutions to use structured data in spreadsheets, it also allows users to filter, transform, and integrate the collected data using familiar spreadsheet mechanisms.

Another popular end-user programming scenario is to program a graphical user interface, such as to create design prototypes [86] or web applications [10,27,38]. Many end-user tools help users create static graphical user interfaces, such as WYSIWYG editors for prototyping (e.g. Balsamic, Adobe Fireworks) and making web pages (e.g., Microsoft FrontPage, Adobe Dreamweaver). Gneiss gives end users the ability to create graphical user interfaces that can interactively use backend data. This is motivated by research showing that end users want to publish their data online and create interactive web pages for browsing and using those data. Benson et al. [9] studied users and web pages created using Exhibit [43], a tool that helps users publish structured data as web pages online (discussed in detail in section 2.3). They found that instead of publishing the data as a shared document or spreadsheet, end users published data as web applications to provide advanced data navigation features, such as facets and visualizations, and to share analysis results in a more understandable way to the audience. Exhibit users were in general frustrated with the traditional web development process and sought a "programming-free" way to author their web applications without writing conventional code.

There are three findings in Benson et al.'s study that further support Gneiss's design: First, while Exhibit lets users create an application that shows spreadsheet data, many end users used JSON data instead because they acquired the data in JSON format or they saw tutorials using JSON to create the visualizations they wanted. Exhibit users could read the JSON data but had much trouble editing it because of the syntax (e.g., dealing with missing brackets). In contrast, Gneiss lets users edit JSON Data in a spreadsheet, eliminating the need to worry about the JSON syntax. Second, they found that the data models that end users used were often more complicated than a table that conventional spreadsheets could support. Gneiss extends spreadsheets to support hierarchical data using nested cells and allows grouping the data by arbitrary fields to create custom, nested data structures. Finally, end users felt that they were often limited by the content management systems they used when publishing data, and sought a way to have complete control over the web pages and data models. In Chapter 4, I will discuss how Gneiss supports creating completely custom web applications including the layouts, the interactions, and the data to use in the backend.

Other studies have shown that non-professional web programmers had trouble building applications that included database functionality. Rosson et al. [75] found that non-professional web developers valued data-related features, such as making forms, surveys and accessing online databases. as much as professional developers did but often were not able to implement them. Voida et al. [85] studied volunteer workers and found that many people used spreadsheets as an informal database for tasks such as recording the RSVP information for a training session since they did not know how to program a database application. In Chapter 6, I will present how Gneiss can turn a spreadsheet into a real database for a web application that lets users store input values entered on a web page.

2.2 MASHUP TOOLS

My dissertation contributes a way to use data from multiple web services and perform custom analyses using spreadsheet mechanisms without writing conventional code. In the literature, applications that make custom use of online data are often called "Mashups." such as applications that combine data from multiple data sources or provide new ways to interact with the data [92]. Many prior mashup tools focused on helping end users get data from the web and connect or combine them without writing conventional code. Many of them use non-spreadsheet approaches. For example, d.mix [38] supports a way to let developers annotate their web pages to enable end users to copy web service calls from those pages and use them in their personal mashups. Marmite [91] and Yahoo Pipes [102] use data-flow languages to let users connect data from multiple sources. Dontcheva et al. [28] lets users extract web page data as "cards" that can have custom layouts and be wired together to specify data flows. DataPalette [49] provides an interface for users to combine similar data from multiple built-in data sources through drag-and-drop and compare the data in visualizations. Those systems are essentially different from Gneiss as they do not use spreadsheets.

Many other prior systems used spreadsheets or tables to help users create mashups. C3W [34] lets users copy web elements from different web pages such as text boxes, drop-down menus and text labels to a single user interface to unify them and specify how data can go from one page to another using a spreadsheet-like language. MashMaker [30] extracts the content of a web page as a tree and lets users modify tree nodes using spreadsheet functions or special widgets such as a filter widget to show certain values and a map widget to plot data on a map. Mashroom [87] lets users get data from web services and web pages and described the structure of the data using nested relational tables, in order to map the data to a GUI template. SpreadATOR [51] supports getting web service data to spreadsheets. It provides a query language to extract desired fields from retrieved documents and lets users define spreadsheet templates on how the extracted data are displayed. Baglietto et al. [5] introduced an architecture that lets users create mashups by linking multiple

spreadsheets and automatically updates the spreadsheets live. Karma [82] allows users to extract data from a web page by example by dragging the first item to a table and letting the system populate the rest of the rows. It also allows the user to simultaneously edit multiple similar cells by example, such as to reformat all phone number cells at once. Vegemite [58] lets users extract data by copying and pasting data from web pages to a table. It further records the user's activities in the browser such as entering text and pressing buttons, to generate step-by-step scripts to reuse in the future.

Gneiss is different from all those tools in many ways. The biggest difference is its ability to create completely customizable and interactive web pages to show data collected from web sources. All the above spreadsheet mashup tools can display mashup data in spreadsheets or tables. Some provide web templates or presentation models to let users map the data to a graphical user interface (e.g., [87]) or show in visualizations. But none of them supports programming custom web layouts, data bindings and interactive behaviors using as this work does. Gneiss supports constructing two-way data flows with web services, which is different from tools that get data from web pages (e.g., [34,58,82]) and tools that only allow retrieving data from but not sending data to web sources (e.g., [30,82,87]). Gneiss also supports using hierarchical documents and restructuring hierarchical data all using spreadsheet mechanisms without having to introduce new query languages (as in [51]) or data models (as in [51,87]).

Some other prior work focused on algorithms to extract data from web pages. Most of them use the structure of the web page and heuristics generated from the characteristics of the page. For example, Dontcheva et al.'s work [29] uses the user's selection and the structure of the web page to extract images and text labels to create summaries of a web page. Sifter [44] extracts search items on a web page using the HTML structure and scrapes subsequent web pages by examining hyperlinks (such as "Next page") and URL parameters. Vispedia [15] extracts Wikipedia infoboxes using the table structure and uses the hyperlinks in an infobox to retrieve related topics. There are also commercial web scrapers, such as Scraper [118], a Chrome plugin for scraping similar items in web pages, ScraperWiki [119], a tool that specifically targets scraping Twitter and tabular data, and Microsoft Excel's Web Query [103] that collects similar data on a web page and puts them into an Excel spreadsheet. Gneiss is different from those tools as it supports using web services and also supports creating data-driven applications, using hierarchical data and streaming data.

2.3 TOOLS FOR CREATING DATA-DRIVEN APPLICATIONS

My dissertation also contributes a novel way to program interactive data-driven web applications – web applications whose main goal is to let people interactively

use data or databases – by using spreadsheet languages. In Gneiss, spreadsheets serve as an intermediate place to hold backend data from different data sources and can be further connected with a web application that can show and even modify the spreadsheet data. The prior work most relevant to this contribution of Gneiss is research on spreadsheet tools that supports programming graphical interfaces that use spreadsheet data. FAR [12] is a spreadsheet tool for creating interactive e-commerce applications. In FAR, the user can create a web page by dragging-and-dropping objects where each object is either a cell or a table. Cells can hold constant values or dynamic values computed using formulas, and the values can be text or images. Tables are groups of cells that share common formulas, similar to a conventional spreadsheet using the autofill gesture to create common formulas in multiple cells. FAR provides a special type of cell called a "query cell" that can take a user's input, use it to query a pre-wired database, and return the query results. Other objects (cells) in the web page can use the query cell's value in formulas to display data from the database.

Gneiss is different from FAR in many ways. First, FAR uses an unconventional spreadsheet interface where cells are objects that can float anywhere in the interface. In contrast, Gneiss adapts a conventional spreadsheet interface that is a table having letters as column labels and numbers as row labels. Second, FAR uses special query cells to get data from a single pre-wired database. In contrast, Gneiss lets users to use multiple, arbitrary REST web services as backend data sources of a web application. Any GUI input elements can be programmed using spreadsheet languages to query a web service or to control how the retrieved data are sorted and filtered. Finally, Gneiss supports programming multi-page applications and interactive behaviors such as interactive visualizations and animations. It is not clear from the publication to what extent FAR could support those things.

Quilt [10] is a JavaScript library developed around the same time as Gneiss that lets people use HTML attributes to connect a DOM element in a web page to data in a Google Spreadsheet. To show data in a spreadsheet, a DOM element can be bound to a single cell (for example, the code `<span connect='B2'></span>` makes the span element show data in spreadsheet cell B2) or be bound to a column in a spreadsheet (in that case, Quilt will create multiple DOM elements based on the number of rows in the column). If an HTML form is bound to a spreadsheet, data flow from the web page to the spreadsheet happens when the form is submitted, which will add a new row of data to the spreadsheet. Gneiss is different from Quilt as it provides a visual environment to create data bindings between a web page and a spreadsheet, whereas Quilt users must create the data bindings by editing HTML files in a text editor. All the data bindings in Gneiss are specified using familiar spreadsheet languages and can be formulas that return dynamic values, whereas in Quilt users specify data bindings using special HTML attributes. Finally, Quilt does not let users

program interactive behaviors using spreadsheet languages, such as the ability to interactively sort, filter and visualize data, which Gneiss supports.

A1 [48] is a spreadsheet environment for programming system administrator tasks such as monitoring network usage. A1 extends spreadsheet cells to be objects that can have different functions, such as queues for storing lists of data, GUI input widgets such as buttons, or objects that connect to external systems such as SSH objects. Like in object-oriented programming, an object (which is a cell in A1) can contain properties and methods. A1 provides a language with a mix of syntax from spreadsheet languages and scripting languages to use the objects (for example, if cell B1 is a queue object, the language `=B1.size()` will return the size of the queue). A1 uses the conventional callback mechanisms to handle events (for example, if B2 is a button and B1 is a queue, the code `on(B2){B1.clear()}` will clear the queue when the button is clicked). Different from A1, Gneiss focuses on creating applications that use backend data sources. Gneiss uses only the spreadsheet language and supports programming interactive behaviors using a pull-based approach that is consistent with the spreadsheet model. It does not require the user to be familiar with scripting language syntax and the event callback mechanism.

Exhibit [43] is a tool that let users publish their data as a web page with faceting and visualization abilities. To use Exhibit, the user starts by embedding the Exhibit JavaScript library and the data she wants to use in the heading of a HTML file. The user can then create web elements by writing HTML code, and turn an element into an Exhibit data browsing widget (such as a control panel that provide options for filtering) or visualization (such as a map or a list to show the data) by giving the element a predefined ID supported by Exhibit (for example, the code `<div id="exhibit-control-panel"></div>` makes the div element a control panel). The user can further use HTML attributes to specify options for the Exhibit object, such as what fields in the data this control panel should provide for faceting.

Exhibit is clearly different from Gneiss as it does not use spreadsheets and does not support using web services or programming interactive behaviors. However, it has a similar motivation as Gneiss, which is to enable end users to create custom web data applications. Exhibit users have more control of the look of the web page as they can freely edit the HTML file, compared with in many content management systems such as WordPress where data are published using premade templates. Gneiss further extends this idea to support creating web data applications with not only custom data and look but also custom interactive behaviors for using the data. And Gneiss provides the ability to program all of them using spreadsheet mechanisms without requiring the user to write HTML or other web programming code.

2.3.1 SPREADSHEET TOOLS FOR MAKING GRAPHICAL USER INTERFACES

Some prior systems extend the spreadsheet model to support programming graphical user interfaces. NoPumpG [89] uses special types of spreadsheet cells to create graphical elements (such as text cells or lines cells) or interactive behaviors (such as a "toggle cell" that respond to click events) in a GUI application. C32 [65] introduces a tabular interface to edit GUI element properties using spreadsheet languages.

Like C32, Penguims [41] also uses a tabular interface for editing GUI element properties using spreadsheet languages. It further lets users construct custom GUI elements using "interactor objects" which are primitive GUI elements provided by the system (such as lines) that have editable style properties such as X and Y coordinates and have some predefined interaction abilities (for example, a line object automatically changes its X and Y coordinates to follow the mouse coordinates when a dragging behavior occurs). Users can combine multiple interactor objects to form a custom GUI object (such as to use four lines to form a rectangle). Besides style properties, a custom GUI object can also have arbitrary numbers of "regular properties" for computation purposes (like variables in a Java object). Like a spreadsheet cell, a property can have a constant value or a dynamic value computed from other properties. Penguims also support object inheritance. The user creates a custom GUI application by combining multiple GUI elements.

Forms/3 [11] is another system that also supports programming custom GUI objects. It uses an unconventional spreadsheet interface where cells can float in the interface based on where the user puts them. Like Penguims, Forms/3 provides a set of graphical primitives (such as circles and lines) whose properties can be edited like spreadsheet cells using spreadsheet languages. It also has a drawing interface that lets users create custom graphics and gestures. The user can combine multiple objects to form a new object. Forms/3 supports programming interactive behaviors by introducing a model of time to the system. The system can record a cell's value and the current time when the cell changes. Considering input events as cells, this essentially allows the system to queue the events based on the time they happened, enabling the user to write spreadsheet formulas that use the most recent events, thus creating custom interactive behaviors. This is different from Penguims where interactive behaviors are not written in spreadsheet languages.

InterState [71] is a recent tool that focuses on programming interactive behaviors using constraints and state machines. It uses a table interface for specifying properties of a GUI element. Each row in a table is a property (such as an X coordinate row and a Y coordinate row), and each column in the table represents a different state (such as a "drag" state and a "no_drag" state when implementing the drag behavior). A property value can be a constant or a constraint which is written using a modified JavaScript syntax. For example, in a "drag" state, the X coordinate

of a GUI element can be `=mouse.X`, and in the "no_drag" state can be `=X` (a self-reference). The transition for one state to another is triggered by events such as mouse down or up.

Gneiss is different from those systems as its central idea is to use the spreadsheet as an intermediate database that can store user inputs and also collect data from web sources as the back end of a GUI application. The systems discussed above do not have a notion of database and do not handle data sources. While Gneiss also allows users to program interactive behaviors using spreadsheet languages, it focuses on supporting data-related interactive behaviors such as interactive sorting, filtering and visualizing data instead of creating general interactive behaviors such as hovering or dragging (although users can program some of them in Gneiss too).

Like Forms/3, Gneiss also has the notion of "time" in the spreadsheet, which is the "cell metadata" that stores the fetched time of a cell having streaming data. But Gneiss only records the fetched time of the current value of a cell and does not allow tracing a cell's previous values by time as Forms/3's time model supports. Again, Gneiss focuses on supporting new ways to use data, and its time model serves as a mechanism to let users select, sort and filter streaming data using temporal information. Whereas Forms/3's time model is for handling GUI I/O and programming custom interactive behaviors, and thus is more complicated. Adding the Forms/3 model to Gneiss might be interesting future work, to enable reasoning about the changes of data from non-streaming sources.

2.3.2 MODEL-VIEW-CONTROLLER LIBRARIES

"Model-view-controller" (MVC) is an architecture pattern popular for programming GUI applications that use backend data [2,55]. It lets developers divide the application into three parts: the "model", which is the data that the application uses; the "view", which is the user interface of the application; and the "controller", which links the model and the view and accepts user inputs in the view to update the model. This partition helps in both maintainability and reusability of the application, as the developer can theoretically modify one part without changing the other two.

MVC is especially popular among web applications [55], and there are many JavaScript libraries designed to support programming MVC applications. Many of them use templating [59,76] to let developers create dynamic web elements generated based on the backend data. A common syntax of a JavaScript template combines HTML tags to specify DOM elements and double curly brackets to specify the data used (such as the name of a JSON object property). Some popular commercial libraries that support templating are AngularJS [104], Handlebar.js [105], Mustache.js [106] and Underscore.js [107]. Both one-way (e.g., React [108]) and multi-way (e.g., AngularJS) constraints are implemented for data bindings. Some

libraries also help developers handle user inputs and program controllers. For example, AngularJS lets developers attach controller functions reacting to a certain input event to a DOM element using HTML attributes. Gneiss is implemented with ConstraintJS [70], a research JavaScript library for creating one-way constraints in web applications. Details on Gneiss's use of constraints for its implementation are described in Chapter 7.

Gneiss contributes a live visual programming environment that supports creating data-driven applications in an MVC fashion but using only spreadsheet languages and interaction techniques. In Gneiss, the model and the view are separated into the spreadsheet and the web interface builder. Data bindings and interactive behaviors are programmed using spreadsheet languages. Like in many templating libraries, web elements in Gneiss can hold dynamic values or be shown and hidden based on the backend data they are bound to. Details on how Gneiss supports programming data-driven applications are described in Chapter 4.

2.3.3 END-USER WEB PROGRAMMING AND VISUALIZATION TOOLS

Web interface builders and WYSIWYG web editors have been widely used in commercial products and research projects to help end-users make web interfaces. However, connecting a web page using a regular interface builder (like Adobe Dreamweaver or Microsoft Visual Studio) to a data source and presenting dynamic content still require writing extensive code. WebSheet [90] and Click [74] try to let users create web applications that use databases in a WYSIWIG editor but are limited to a few simple built-in database actions supported in the system such as adding a new row (both tools also do not appear to be fully developed). Some other research tools help people use examples to create static styles [18,53] or interactive behaviors [72] of a web page. But they do not help users understand or reproduce how the example web pages use and interact with the backend data sources.

Many conventional spreadsheet tools (such as Google Spreadsheets and Microsoft Excel) and visualization tools (such as Tableau [80] and IBM ManyEyes [84]) support creating interactive visualizations that use spreadsheet or table data. But those tools do not support building a web application where the dataflows between the web interface and the spreadsheet are bi-directional.

2.4 TOOLS FOR USING HIERARCHICAL DATA

Gneiss extends spreadsheets to support using structural hierarchical data formats such as JSON data. The goal is to enable end users to work with those formats of data, as these data are becoming more and more popular due to the increasing amount of web services and web applications. There are already many libraries that help professional programmers use hierarchical data, such as JSON.simple [109],

GSON [110], FasterXML's Jackson project [111], Jansson [112] and jsonQ [113] for JSON data. In this section, I focus on reviewing prior tools that help end-users work with hierarchical data.

In 1991, Lotus Improv introduced the concept of "categories" that contain related attributes of the spreadsheet data. The user can create custom categories, such as a "season" category that has "spring", "summer", "fall" and "winter, and a "year" category that has a list of years. The user can then use these categories as row and column labels to reshape the data, such as to create a plot using years as column labels and seasons as row labels. This feature was eventually replaced by pivot tables in today's spreadsheets, where the user can perform different groupings on data and calculate summaries such as counting the items in each group.

Although a pivot table allows users to create nested groups, it assumes the underlying data are flat and do not contain multiple nested structures, which in hierarchical data are very common. For example, a movie database could return a list of movies where each movie has a list of actors, a list of reviews, and a list of awards. Turning such data into a flat table will either create lots of repetitive rows or lots of columns with many empty cells (depending on how the lists are expanded), as each movie can have different numbers of actors, reviews and awards. Some tools support transforming table data into long and wide format [37,47,73]. However, our user study showed that transforming a hierarchical document into either format would still require users to do additional data manipulation before using pivot tables. There are data cleaning tools that let users quickly edit strings, such as Excel's Flash Fill [36] and Wrangler [47]. Those tools also do not help much when using hierarchical files, as the problem is with the data structures rather than the string values.

A lot of work focuses on visualizing hierarchical data using tables and graphs. Gneiss is inspired by prior work on using nested tables to visualize nested relational models (see [56] for an overview). In a nested relational model, the nested table often has a nested heading showing the schema that may contain attributes and sub-relations, and a nested content area showing the instances of the model. Many systems were built based on the nested relational model to support basic editing and querying databases [88] and visualizing hierarchical documents to increase readability [8,26]. Compared with those systems, Gneiss uses a different method to turn hierarchical data into nested tables that focuses on supporting *reshaping and regrouping data*. Gneiss' visualization method allows users to dynamically create different views of the same hierarchical object using interaction techniques, with the goal of facilitating data exploration. This is different from prior systems that generate a single static view for a hierarchical object.

Related Worksheets [7] lets users create inner cells in a row to show one-to-many relationships in data. A cell can be set to reference values from another worksheet and thus may create additional nested cells. Their study showed that users could understand the nested cells and could use them to find information. However, the Related Worksheets does not support further manipulating the nested data, such as sorting, filtering or calculating new data with functions like in our tool. The nested cells serve only for viewing purposes. This is another difference of this work and prior research. Gneiss' hierarchical visualization is designed to enable users to reference nested cells (and even select them using their hierarchies) to use in formulas using the familiar spreadsheet syntax (the combination of column label and row number). Another difference is that many prior systems (such as [8,26]) did not address how data in the complex nested tables generated by their systems could be further manipulated by the users.

There are spreadsheet tools that use nested cells or linked spreadsheets to represent hierarchies in other programming activities, such as defining new spreadsheet functions and parameters in functions [46], creating spreadsheet templates [1], specifying data models [87], and programming graphical interfaces and object inheritance [11]. However, those systems do not support exploring and manipulating hierarchical datasets as our tool does.

Some other research tools focus on extracting hierarchical relationships in conventional spreadsheets created implicitly by users. For example, Hermans et al. used content layout and cell dependencies to extract hierarchical information in spreadsheets and visualize it as diagrams to improve spreadsheet readability [39,40]. Chen et al. used fonts and alignments to extract hierarchical information in a spreadsheet to generate relational tables which can be used in relational databases [23,24]. Different from those tools, our work targets hierarchical data formats such as JSON and XML that are not designed for relational databases, and the hierarchies in the data are explicitly defined. Our tool directly operates on hierarchical objects (instead of flattening them into tables) and focuses on leveraging those objects' structural information to facilitate data manipulations such as to support hierarchical grouping, sorting and filtering.

2.5 TOOLS FOR USING STREAMING DATA

Finally, Gneiss contributes a way to use streaming data in spreadsheets. Developed around the same time as Gneiss, ActiveSheets [83] is a spreadsheet tool built on top of Excel that supports live streaming data. ActiveSheets supports many similar features for using streaming data in a spreadsheet as Gneiss, such as the ability to have live data and live computation in a spreadsheet, to be able to filter data, and to preserve history values. The main difference between ActiveSheets and Gneiss is in the interactions. ActiveSheets support most of its features using text-based

approaches. For example, creating a stream is done by clicking on an icon and entering a stream name and a window size (for how many values to display in the spreadsheet). Filtering a stream is done by a spreadsheet function. In contrast, Gneiss uses more interaction techniques to achieve those things. For example, creating a stream in Gneiss is done by dragging-and-dropping a field from the source pane to a spreadsheet column. Filtering a stream is done using the sorting and filtering widget as in conventional spreadsheets. Gneiss supports streaming data from arbitrary REST web services, whereas ActiveSheets uses pre-wired streaming data sources on a dedicated server. The two tools also have different ways to manipulate streaming data using temporal information. ActiveSheets has a focus on generating new streams using spreadsheets. It allows users to produce a stream calculated based on values from the original streaming source and export a stream to share with other spreadsheet clients. Gneiss does not support those features. Instead, Gneiss focuses on creating web applications and can let users create interactive web applications that use and control data streams (such as pausing a stream or changing streaming frequency). While ActiveSheets can let users create live visualizations, it does not support creating web applications.

Woo et al. [93] introduce an architecture for using continuous sensor data in spreadsheets. This work focuses strictly on sensor data and is quite different from a general-purpose tool like Gneiss that supports arbitrary data sources. There are also many commercial tools that let users use and analyze streaming data in spreadsheets. Most of the commercial tools use pre-wired data sources on the server and many of them provide analytics ability. Examples are StreamBase [98], IBM's Streaming Analytics Service and InfoSphere Streams for Excel [114], and Microsoft's Azure Stream Analytics for PowerBI [115].

There are many professional languages and libraries that help programmers work with streaming data by writing code. For example, StreamIt [35] is a language for processing streaming data on multicore processors. Spark Streaming is a library that supports analyzing streaming data using the Spark engine [94]. Some work leverages SQL techniques to support streaming data, such as Microsoft StreamInsight [3] and PipelineDB [116] that extend SQL to support querying streaming data over time windows and return live results. Some other systems support both querying streaming data and creating visualizations to help programmers understand the data. For example, Tempe [32] is a live programming environment for data scientists to query streaming data and view and interact with the results in both tables and graphs. Kibana [97] is a visualization tool that allows developers to create live visualizations of streaming data. All these tools target users who are sophisticated programmers.

2.6 CONCLUSIONS

This research is motivated by many use cases described in prior literature where end users needed to work with web sources and databases and publish data online. Gneiss extends the spreadsheet model, aiming to leverage its familiar and direct programming language and its natural live computing model to make using web services and creating data-driven applications easier for people familiar with spreadsheets. I have discussed many related systems and showed how Gneiss makes unique contributions in comparison. Next I will describe those contributions in detail, starting by introducing how Gneiss supports programming two-way data communications with arbitrary REST web services using spreadsheet mechanisms.

CHAPTER 3 USING WEB DATA SERVICES[4]

Today, many data sources provide web services that allow people to access their data programmatically. Some web services not only provide data but also computational services on data, such as transforming geo locations to country codes (e.g., GeoNames), or running machine learning algorithms and storing the data sent by the user (e.g., Amazon). Web services can be powerful resources for people who have the need to use online data in custom ways. However, using web services currently requires writing a significant amount of surprisingly intricate code that deals with asynchronous network calls which may fail to return, often requiring complex and sometimes nested callbacks [70,95].

3.1 MOTIVATION, CHALLENGES AND CONTRIBUTIONS

As discussed in Chapter 2.1.1, spreadsheets are the most popular end-user data tools and have many familiar functions and interaction techniques for manipulating data. Conventional spreadsheet systems support using data from files, databases, and some even support scraping data from web pages (such as Excel's Power Query [103]). But when it comes to using web services, current spreadsheet systems fall short in several ways:

- Some conventional spreadsheets provide general functions that let users get data from an arbitrary web service given a web URL, such as Excel's `WEBSERVICE(url)` function. However, these functions return the entire retrieved document as a plain text string and put it in a single cell. Thus the data becomes difficult to read, select and manipulate. Those functions also do not support sending data to a web service.
- Some spreadsheets provide functions that get data from or send data to specific data sources. For example, Google Spreadsheets have a `GOOGLEFINANCE` function with which the user can get finance data for a company at a certain time using parameters to the function. Those functions often return more readable and usable data than functions that let people get data from arbitrary web services, but creating a new function like those to use a new data source would require programming in non-spreadsheet languages, such as JavaScript for Google Spreadsheets or VBA for Excel.
- In conventional spreadsheets, since web service data are retrieved dynamically using formulas, they cannot be sorted and filtered in conventional spreadsheets, as sorting and filtering do not work on formula data. The user would need to copy the data and paste them as constants to perform further manipulations.

[4] The research in this chapter was also described in our publication at VL/HCC'14 [20].

- The majority of modern web services today return hierarchical documents such as JSON and XML data. However, current spreadsheets only support table data. Hierarchical documents need to be flattened first before used in a spreadsheet. This removes much structural information in the data that could be useful for analyzing and manipulating the data.
- Some web services provide real-time data, such as finance data or geo-locations of people and vehicles. Currently, some spreadsheet plugin tools support streaming live data in spreadsheets. However, those tools require data sources to be hardwired into the system. Adding a new data sources is often impossible for end users.

My dissertation extends the spreadsheet model to address these problems. In this chapter, I focus on describing Gneiss's contributions to address the first three bullet points. Contributions on the last two bullet points about how Gneiss supports hierarchical data and streaming data are described respectively in Chapters 5 and 6.

Specifically, in this chapter I present the following contributions:

- Gneiss's spreadsheet language and interaction technique to send data to and retrieve data from arbitrary REST JSON web services without writing conventional code.
- Extensions to spreadsheet's sorting and filtering to apply persistent sorting and filtering rules on web service data.
- Extensions to spreadsheet's autofilling gesture to support sending similar web service requests by example.
- Gneiss's nested cell visualization to show hierarchical data, and the spreadsheet language to select the data by its structure.
- A way to refresh web service data in spreadsheets.
- How the spreadsheet model naturally handles different states of an asynchronous web service call including errors, and creates parallel-running data extraction programs based on the user's sequential demonstration.

3.2 INTERFACE OVERVIEW

Gneiss contains three panes (see Figure 3.1): a "source pane" (left) for where the user can load data from a web service by entering a web API in the URL bar at the top or load a local file using the "Choose file" button; a spreadsheet editor (center); and a web interface builder (right) where the user can create a web application by dragging-and-dropping GUI elements from the side bar at the far right to web pages and editing the properties of elements in the property sheets at the lower right corner (not shown) The user can choose to hide the source pane and the web interface builder if she does not need them. This three-pane design is based on many conventional code editors where the file directories are put at the left of the

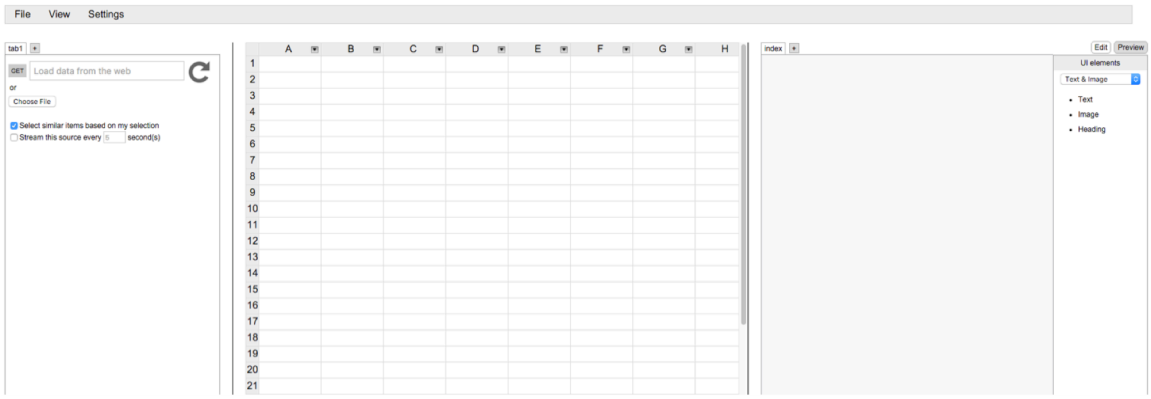


Figure 3.1. A screenshot of Gneiss's interface.

source code (such as in Eclipse), and the graphical view of textual code is put at the right of the source code (such as in Adobe Dreamweaver).

Gneiss's source pane uses a browser-like design. The user can load an external data source by entering a web API to the top URL bar, or by clicking the "Choose file" button to use a local file. The user can click on the "+" icon next to the rightmost tab to open a new tab to use multiple data sources in Gneiss.

To allow myself the required flexibility, I implemented my own spreadsheet and web interface builder. Gneiss's spreadsheet looks and works similar to conventional spreadsheet editors. The user can double-click on a cell to edit it. On each column's heading, there is a small arrow icon on which the user can click to open a dialog box to sort and filer data by that column. Gneiss currently supports a limited set of spreadsheet functions. The complete list of supported functions is in Appendix A. Currently Gneiss's spreadsheet has 50 rows. If the data exceed 50 rows, the spreadsheet will show the first 50 rows of data and hide the rest of the rows for performance reasons. All the data operations such as sorting, filtering, grouping and joining are executed on the entire dataset. Similarly, for autofill, as long as the user drags down to the last displayed row, the system will apply the operation to the entire dataset including the hidden rows. To view the hidden data, the user can use sorting and filtering to bring the data she wants to the top, or open the entire data in a read-only spreadsheet using a menu command. Many conventional tools have similar design. For example, Google Fusion Tables show 100 rows at a time and have a "next" button to let the user view the next 100 rows of data.

Gneiss's web interface builder lets users create a UI element in web page by drag-and-dropping an element from the side bar to the page. The UI elements are organized into categories. The user can select a category using a dropdown menu and view a set of available UI elements (for example, in Figure 3.1, the currently category is "Text & Image", and there are three types of UI elements available, which are "Text", "Image" and "Heading"). Gneiss also provides a set of visualizations in the

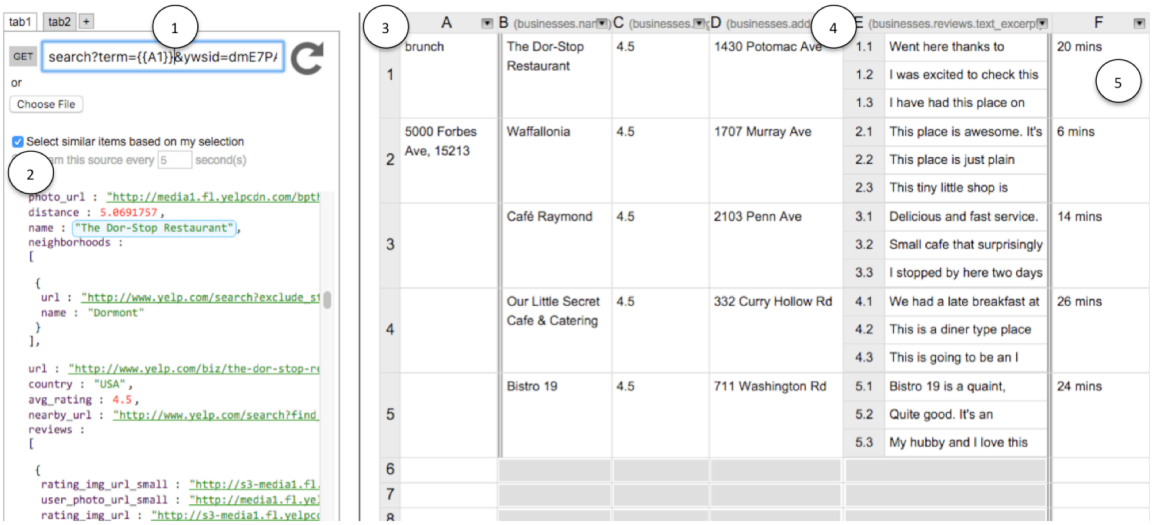


Figure 3.2. A screenshot of Gneiss showing the spreadsheet program created in the usage scenario. At the left is a source pane that shows the raw data returned from a web service. (1) is the URL textbox where the user enters the address of a web API. Note that the value of the cell A1 has been used as the search term. (2) is the returned data. (3) is the spreadsheet interface where the user can store desired fields extracted from the raw web service data and do manipulations. (4) If the extracted data have structure, they are shown in nested tables. The final results, the driving time from Alice's school to the restaurant, are shown in column F at (5).

web interface builder using Google's Visualization API. The complete lists of currently supported web UI elements and visualizations are in Appendix B. The web interface builder has an "Edit" mode and a "Preview" mode (selected by pressing the "Edit" and "Preview" buttons at the upper right). In edit mode, the user can drags UI elements from the side bar, click on a UI element and edit its properties in a property sheet (shown in Figure 4.1 at 5 in chapter 4). In preview mode, the system hides the side bar and lets users test the created application as if it was opened in a regular web browser. Similar to the source pane, the user can click on the "+" icon next to the rightmost tab to open a new page to create a multi-page application.

3.3 USAGE SCENARIO

Here I describe a usage scenario to give an overview of how Gneiss enables users to create spreadsheet programs that integrate data from multiple web services.
In this scenario, Alice uses Gneiss to create a spreadsheet program that uses a restaurant web service to look for the highest rated restaurants and then uses a direction web service to calculate the driving time from her school to the restaurant. Alice is a fluent spreadsheet user familiar with sorting, filtering and using functions in spreadsheets. She also knows the web APIs that she can use to get the information about restaurants and directions. This task (finding places in one data source and calculating the route to the place in another) is a popular task frequently used in prior literature (e.g., [28,91]). Figure 2 shows a screenshot of the final result (the web interface builder is not used in this scenario and thus is hidden.)

Alice starts by searching for a web service for restaurants online. She quickly finds that Yelp provides a restaurant query API that returns a list of restaurants given a search term. Alice copies the query API from Yelp's documentation page and pastes it into the URL textbox (Figure 3.2 at 1) in Gneiss's source pane. The return data are shown below the textbox (Figure 3.2 at 2). Alice puts the search term in cell A1 and sends it to the web services to retrieve data. To do so, she changes the value for "term" to be `{{A1}}` (Figure 3.2 at 1). Now every time that A1 changes, a new restaurant search request is sent using A1's value as the query string, and the source pane updates to show the latest return data.

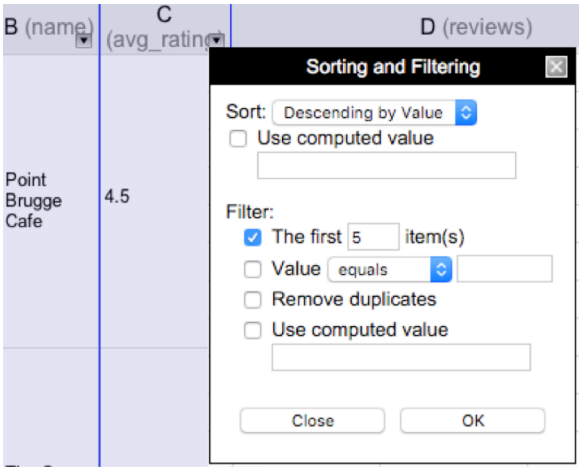


Figure 3.3. Pressing the arrow icon at the top of each column brings up a dialog box that lets the user apply sorting and filtering to that column. Data extracted from the same source (in neighboring columns) are sorted and filtered together and are highlighted with a purple background. The column from which the dialog box is opened is highlighted with a purple border.

Alice does a few tests and makes sure this part works correctly by trying different search strings in A1. She then starts to extract the fields she wants. Alice first wants the name of each restaurant. She clicks on the "name" field of the first restaurant. The field gets highlighted with a blue background. She then drags the field and drops it on column B in the spreadsheet. The tool automatically extracts the names of all restaurants and puts them in column B, and a grey label appears in column B's label to show the field name from where the data was extracted. Alice then extracts the rating, address and reviews of each restaurant in the same way. The API returns multiple reviews for a restaurant. Alice selects the first review text of the first restaurant and drops it on column E. The tool populates the rest of the cells in the column and shows the reviews in nested tables (Figure 3.2 at 4).

Alice only wants to view the top 5 rated restaurants. To do so, she clicks on the arrow button at the top of column C, the column that stores the ratings, to bring up a dialog box that lets her sort and filter the extracted data (Figure 3.3). She sorts the column descending and filters to show only the first five items. She does a few more tests by editing A1 to search for a new kinds of restaurants. Changing A1 triggers a new request using A1 as the query term to be sent to the web service. When the request returns, the spreadsheet updates with the latest search results and shows a new list of restaurant names, addresses, ratings and reviews. The sorting and

filtering rules are re-executed dynamically every time new data arrive, so the spreadsheet will always show the top 5 rated restaurants.

The last step is to use a direction web service to calculate the time from Alice's school to the restaurants. Alice enters Google's direction API in the URL bar, and she binds the value of parameter "origin" and "destination" of the API to the value of cell A2, which has the address of her school, and D1, the address of the first restaurant. The API returns the time, and she drags-and-drops it into cell F1, while F2 to F5 stay empty. To send four more direction requests using the other four restaurants' addresses as destinations to fill in F2 to F5, Alice selects F1 and moves the mouse to its bottom-right corner. The mouse becomes a "plus" sign for the familiar "auto-fill" command. She presses the mouse and drags down to F5 to fill in F2 to F5 with the time-to-destination of the other restaurants (Figure 3.2 at 5).

Alice has now finished her data extraction program that uses Yelp's and Google's web services to find top-rated restaurants and the driving time from her school to the restaurants! She can easily look for another type of restaurants by changing A1, or instead view the time-to-destination from her house to the restaurants by changing A2. The spreadsheet will update to show the results based on the latest values in A1 and A2. While the direction requests can only be sent after the restaurant search request returns so all the addresses are filled in, among the direction requests there are no dependencies. Therefore, the five direction requests are sent in parallel to speed up the performance. Alice saves the spreadsheet she just created. The next time when she needs to do another restaurant query, she only has to load the spreadsheet back into the tool.

3.4 KEY FEATURES FOR USING WEB SERVICES

In this section I describe the key features in Gneiss's spreadsheet model to assist people in using web services.

3.4.1 SELECTING AND EXTRACTING WEB SERVICE DATA

In Gneiss, raw data returned from web services are shown in the source pane at the left of the interface. Gneiss supports arbitrary REST web services that return JSON data (extending Gneiss to handle other formats such as XML would be straightforward but is left for future work. See also section 3.6). The returned data is formatted in the source pane to increase readability. To extract a desired field, the user first selects the field by clicking on it. In the source pane, there is a "select similar fields" checkbox (Figure 3.2 at 2). If the checkbox is checked, the system will select other similar fields using the user's selection as an example. To select similar fields, the system will recursively go through other items in the same array as the selected field and collect fields that have the same path. For example, in the usage

scenario, the user selects the first review of the first restaurant (`$businesses[0].reviews[0].text_excerpt`). With the "select similar fields" checkbox checked, the system will select all reviews of all restaurants (`$businesses[*].reviews[*].text_excerpt`) for the user. The selected fields will be highlighted in a blue background in the source pane. If the "select similar fields" checkbox is not checked, the system will only select the field that the user clicks on.

As described in the usage scenario, the user can extract selected fields to the spreadsheet editor by drag-and-drop. If the "select similar fields" checkbox is checked, the user can drag selected fields to a spreadsheet column, and the column gets reserved to only show those data. Empty cells in the column are greyed out to show that they are not available for manual edits. Since the web service data come in dynamically, we adopted this design to avoid the situation where the user's data gets accidentally erased if the new web service data is longer than the first demonstration. For example, in the scenario in section 3.3, the number of restaurants returned from the web service might be different each time based on different query terms. Therefore, the system greys out cells in column B-E that do not have data (see Figure 3.2, column B – E from row 6 and below).

To remove columns that contain web service data in the spreadsheet, the user can select a column and choose to clear all contents to start over. Users are free to type anywhere in columns that just contain user-typed data (such as column A in the usage scenario). If the "select similar fields" checkbox is not checked, then only the selected field is used in the target spreadsheet cell. Other cells in that column are not reserved and can be edited like regular cells.

In an earlier version of Gneiss [20], after selecting a field, the system would display the path of the selected fields in the source pane and allow the user to edit the path to perform more customized selection using the data structure. For example, in the usage scenario, if the user only wants the first review of each restaurant, she could first click on the first review of the first restaurant, and edit the path to be `$businesses[*].reviews[0].text_excerpt` to change the selection, then drag the selected fields to a spreadsheet column. This feature was removed in the current version [22] as the user can perform hierarchical sorting and filtering in the spreadsheet to achieve the same thing without having to understand the JSON path syntax. For example, the user can now first extract all restaurant names and all their reviews then use filtering to keep only the first review of each restaurant. Gneiss's support for manipulating hierarchical data is discussed in detail in Chapter 5.

3.4.2 SENDING SPREADSHEET DATA TO WEB SERVICES

As described in the usage scenario, the user can send data in a spreadsheet cell to a web service by embedding the cell name in a web API using the syntax

`{{cellName}}`. The double braces syntax is adapted from conventional web template libraries such as Handlebar.js [105] that let the programmer escape from HTML and write JavaScript statements inside the braces. Here, I use this syntax to let users write spreadsheet expressions in web API URLs. As I showed in the scenario in section 3.3, the spreadsheet data used in web APIs can either be constant (like A1) or computed based on other cells (like E1). RESTful web APIs are typically HTTP requests. Gneiss by default sends an API request as a GET request. The user can change it to a POST request when using a POST API by toggling a button next to the URL bar in the source pane (Figure 3.2 at the upper left).

Gneiss uses the spreadsheet's one-way constraint evaluation model to send a web service call and handle the return data. As demonstrated in the usage scenario, if a web service request uses values from spreadsheet cells, every time when the cells change, it will trigger the system to resend the request using the new values. When waiting for the request to return, spreadsheet cells that contain data extracted from that web service request will become a special "Loading…" value (similar to how conventional spreadsheets handle errors). The "Loading…" value will propagate throughout the spreadsheet to other cells that use data from this web service in formulas. When the web service request returns, all corresponding spreadsheet cells will automatically recalculate using the latest return data. If a spreadsheet cell whose value comes from a web service call that depends on another spreadsheet cell that is currently loading, the system will not send the web service call until the dependent cell receives its value. For example, in the scenario in section 3.3, when the user searches for a new type of restaurant, cell F1's web service call (driving time from cell A2 to cell D1) will not be sent until the restaurant search results (column B-E) finish loading, and F1's value is "Loading…" while waiting for D1. If a web service call fails to return, the system will populate an "Error" value in all the related cells. The users do not need to write any code to deal with different states of an asynchronous network call. As in conventional spreadsheets, errors are localized and will not affect other independent spreadsheet cells.

The drag-and-drop gesture for extracting web service data to spreadsheets and the double-braces syntax for sending spreadsheet data to a web service achieve two-way data flow between Gneiss and arbitrary REST web services. The spreadsheets created in Gneiss are easily reusable, as the user can send a new request to a web service and retrieve new data by simply editing spreadsheet cells. The spreadsheet can further be linked to a web interface created in Gneiss's web interface builder (Figure 3.1 at the right) to let people search and view the data in a web application (described in detail in Chapter 4).

3.4.3 SORTING AND FILTERING WEB SERVICE DATA

Computed Rules for Sorting	
"None", "Ascending", "Descending", "Ascending by time", "Descending by time" (the last two for streaming data only – see Chapter 6)	
Computed Rules for Filtering	
Method Name	**Parameter(s)**
"Filter top"	The number of first X items to keep
"Filter value"	1) "=", "contains", ">=", "<=", ">", "<" 2) The value to filter
"Filter duplicates"	No parameters
"Filter by time" (for streaming data only)	1) "Before", "After" 2) The time to filter

Table 3.1. Computed rules for sorting and filtering supported in Gneiss.

As described in the scenario in section 3.3, sorting and filtering rules specified for columns that have web service data are re-executed every time when the system retrieves new data from the web service to maintain the relationship. This provides a way to let users further refine the collected external data, such as in the scenario in section 3.3, Alice can use sorting and filtering to always view the top 5 rated restaurants every time she makes a new query. This is different from the regular sorting and filtering in conventional spreadsheets that only sort and filter the current data in a column and do not apply to future edits. In fact, in conventional spreadsheets, if web service data are retrieved by functions, they cannot even be sorted and filtered because sorting and filtering do not work on data generated by spreadsheet formulas. My design in Gneiss avoids this problem since web service data are retrieved through the drag-and-drop gesture.

Currently, Gneiss supports sorting web service data in a column by the values. If the column stores streaming data from a web service, the user can also choose to sort the data by its fetched time (it does not make sense to sort non-streaming web service data in a column by fetched time since they are all retrieved at the same time, such as the restaurant search results in the scenario in section 3.3). Details on sorting streaming data by time are described in Chapter 6.

For filtering, Gneiss supports filtering to view the first X elements (as in the scenario section 3.3 to view the top 5 rated restaurants); to view data that are equal to, less than, greater than, less than or equal to, greater than or equal to, or contain (for strings only) a certain value; or to remove all duplicate values. If the column stores streaming data from a web service, the user can also choose to view data within a certain time period.

To apply sorting and filtering to a column, the user clicks on the arrow icon at the bottom-right corner of the column title (Figure 3.3). A dialog box will appear to let the user specify how they want the column to be sorted and filtered. Currently, adjacent columns extracted from the same web service request are sorted and filtered together. For example, in the scenario in section 3.3, the user opened the dialog box for the rating column (column C) and applied sorting, and that sorts the entire restaurant data in column B-E by column C. A future work is to let users select

the affected columns in a dialog box as in conventional spreadsheets. When the sorting and filtering dialog box is open, Gneiss highlights all the affected columns using a purple background, with the column to which the dialog box belongs having a purple border to make this clearer to the user.

Sorting and filtering rules can be constant rules set using the GUI controls in the dialog box as in the scenario in section 3.3. They can also be dynamically computed using a spreadsheet formula entered into the "Use computed value" textbox in the dialog box. For sorting, the computed value should be a string that describes the sorting method. For filtering, the computed value should be a comma-separated string with the first item being the filtering method and the rest of the items being the parameters required by the method (see **Error! Reference source not found.** for a complete reference). For example, entering `=IF(A3>1, "Descending", "Ascending")` in the "Use computed value" textbox for sorting will sort the data descending if A1 is bigger than 1 and ascending otherwise. This feature is mainly designed for letting users create web applications that can sort and filter spreadsheet data using web GUI element, such as by entering `="Filter value, >=, "&Slider1!Value` as the rule to filter out cells that are less than the value of `Slider1` in the web application. I will explain the details about building web applications in Gneiss in the next chapter (Chapter 4).

To sum up, Gneiss provides two levels of dynamic sorting and filtering for web service data. First, sorting and filtering rules are re-executed every time when new data are retrieved from a web service, so that the data is always in the desired order and value range. Second, if the sorting and filtering rules are computed values, every time when the computed values change it will cause sorting and filtering to be run again as well.

3.4.4 "AUTOFILLING" CELLS WITH WEB SERVICE DATA

"Autofill" (also called "fill down") is a common feature in spreadsheet tools where the user selects one or multiple cells and drags to fill in additional cells. In Gneiss, when the user selects and drags a cell whose value comes from a web API that uses values of other cells in the same row, as the user drags it down, the system will replace these cells in the web API with the corresponding cells in the new row. For example, in the scenario in section 3.3, cell F1's value comes from the direction API that uses the value of cell A2 and E1 as the value of the origin and destination parameter (Figure 3.2 at 5). When the user selects F1, the system recognizes that E1 is in the same row as F1. When the user drags down to rows 2 and 3, the system replaces E1's value in the web API with E2's and E3's values (the addresses of the second and third restaurants), sending the required API requests, and extracting data using the same path as data in F1 to fill in F2 and F3. By default, our tool does not replace A2's value in the web API with A3 and A4 because A2 is not in the same

row as the selected cell E1. If the user wants A2's value to also be replaced by the rest of the cells in column A, she can manually compose a second cell using another direction API call that uses the value of A3 and E2, and extracts the data to F2. Then, if she selects both F1 and F2 and drags, the system will recognize the pattern in the selected cells and apply it to fill in the new cells. This is the same way that Excel and other spreadsheets work now – if the autofill pattern is not apparent from a single cell, users can provide two cells to demonstrate the pattern.

As described earlier, a web service request could return different numbers of data depending on the query terms. In Gneiss, when the user selects a cell whose value comes from a web API that uses values of other cells in the same row and drags to the last row of the current column, the system will reserve this column (empty rows are greyed out) and autofill it to have the same number of items as the example columns. Of course, the user can drag it to a specific row if she knows exactly how many calls to send. For example, in the scenario in section 3.3, the user filters to only have just the first five restaurants, so when autofilling column F with direction web service calls, she knows that she only needs to drag to F5. In this case, column F will not be reserved, and the user can use the rest of the cells in F for other things. This can also improve performance at run time, as the distance web service only needs to be called 5 times.

3.4.5 USING STRUCTURED DATA

When dragging web service data to the spreadsheet, Gneiss will display data as nested tables to show the hierarchical structure among the extracted data. For example, in the usage scenario, when the user extracts the reviews to the spreadsheet, because each restaurant can have multiple reviews, the system generates nested tables to put the reviews in the same row with their corresponding restaurant. Each nested cell in a nested table has a nested row label that can be used to reference a cell in a spreadsheet formula using the familiar syntax `ColumnlabelRowlabel`. For example, in Figure 3.2 at 4, `E1.1` selects the value "Went here thanks to…" which is the first review of the first restaurant. The user can also uses conventional spreadsheet's ":" operator for specifying the start and end cell of a range selection to select values in multiple cells. For example, `E1.1:E2.2` in Figure 2 at 4 returns five reviews. The user can also use the parent row label to select all cells in a nested table. For example, `E1` returns all three reviews of the first restaurant in Figure 3.2 at 4. Finally, our language includes a wildcard character (`*`) that can be used in any nested row index to further assist hierarchical selection. For example, `E*.1` in Figure 3.2 at 4 returns the first review cell of all the restaurants.

Values returned by a selection are put in a one-level array; in other words, the selected nested cells are flattened and put in a flat array. For example, in the spreadsheet in Figure 3.2, `E1` returns an array of three items with each item being a

review. Therefore, conventional spreadsheet functions that accept a list of values can also work for nested cells. For example, the user can write this formula `=COUNTIF(E1, "mussels")` to use the conventional `COUNTIF` function to count the number of reviews of the first restaurant that contain the word "mussels".

This extended spreadsheet language syntax leverages the structure of the data to support selecting hierarchical data and using the data in spreadsheet formulas. In fact, Gneiss' hierarchical data visualization is dynamic based on how the user organizes data in spreadsheet columns and allows the user to regroup hierarchical data using arbitrary fields. The user can also sort, filter and join hierarchical objects using their hierarchical structure. The complete features for using hierarchical data in Gneiss are described in detail in Chapter 5.

3.4.6 REFRESHING WEB SERVICE DATA

Many web services provide dynamic data that change over time. For example, in the usage scenario, for the same destination the Google web service may return different driving times in different hours of a day. If the spreadsheet has been created for a while, the user may want to update the spreadsheet with the latest values.

Gneiss provides two methods to let users "refresh" spreadsheets data from web services. First, as mentioned in the introduction, Gneiss supports using streaming data in spreadsheet by periodically pulling data from a web service and stacking the data in a spreadsheet column. This creates a "growing column" where history values are stored in the column and new data are added to the top of the column periodically. The streaming data feature is good for users who want to do analysis that involve historical data, for example, to see the trends of the stock market in the last 5 days. Details on how Gneiss supports streaming data are described in Chapter 6.

Gneiss provides a second method to refresh spreadsheet data using a dialog box in a menu that will resend all web service requests used by the spreadsheet and then update the spreadsheet with the latest return data. This second method is different from the streaming data method as it does not store historical data but simply pulls the data from web services again and replaces cell values with the new data. If the user only cares about the current data, such as in the usage scenario where Alice only wants to know how long it would take to drive to the restaurants *now*, then using the second method is sufficient. If the user cares about not only the current data but also the trends in the data or wants to analyze the data by time (e.g., if the user wants to analyze the traffic jam situation throughout a day), then she will use the streaming data feature.

3.4.7 PARALLEL-RUNNING PROGRAMS, ERROR HANDLING AND MAINTENANCE

While the user's demonstration and manipulation of cells is always performed sequentially, the spreadsheet metaphor allows Gneiss to construct a parallel-running program using the dependencies among the spreadsheet cells. Cells that do not have any dependencies on each other can be computed independently in parallel. This could make a big improvement on performance, especially when extracting a large amount of data. For example, if the user wants to collect data using 50 on-line shopping web services to compare prices, she can easily create a spreadsheet program that sends 50 web API requests in parallel, and gets the data within seconds. This is in contrast to web scraping programs (e.g., [30,58,82]) that can only execute sequentially in the same order as when the user demonstrated them. Those systems also must insert fairly long delays into the program (for example, 10 seconds in the example in [58]) in hopes that this will be long enough to make sure the web pages have finished loading. Since our tool uses web services, we know when the call has completed, and no extra delays are required.

A web service call could sometimes fail due to various reasons such as scheduled maintenance, change of protocols, a bad Internet connection, or even a bug in the user's specification of the URL. As briefly discussed in section 3.4.1, Gneiss is robust in handling bad web service requests, as it will fill the error cells with a special "error" value and propagate this value to all dependent cells when it receives a HTTP error when accessing the web service. The user can easily see the errors in the spreadsheet and trace back to the origin using conventional spreadsheet mechanisms. Our system's ability to execute programs in parallel also allows other parts of the program that do not depend on the error cells to run as usual. For example, if the user creates a spreadsheet that searches a product on 10 different web services and the request to one of them fails, the user will see an error from that web service but still will still get data from the other 9. Debugging is also easier in our tool because the user can see all the values in the spreadsheet and see them changing at run time, unlike in other programming languages, such as Yahoo Pipes, where data are typically hidden unless the user looks for them.

As described in the scenario, the user can save the spreadsheet program and load it back to the tool at a future time when she needs to perform similar queries. Reusing a spreadsheet is easy as the user can edit a spreadsheet into another one and reuse appropriate parts. For example, in our scenario, if Yelp stops supporting its web service after the spreadsheet program is created, the user could use another web service to search for restaurants (such as Google Places) and replace the restaurant columns in the same spreadsheet with appropriate fields from the new data source, without having to modify the location part.

3.5 DEMONSTRATIVE EXAMPLES

Here I use two more examples to demonstrate Gneiss's ability to create spreadsheet programs that use web services in a variety of ways.

3.5.1 CITY TRIP PLANNER WITH A MAP

A common activity when a person plans to visit a new city is to search for all attractions in the city, pick the ones he is interested in, and plot them on a map. In this example, I will show how a user with basic knowledge of spreadsheet programming can program such a custom application with Gneiss using spreadsheet languages.

The user starts by using Google's Place Search API to get a list of attractions. She changes the value of the query parameter in the API to `{{A1}}` to bind it to cell A1. From the return data, she drags the name, rating, latitude and longitude fields to columns B to E. She decides to make column F the "input column" – if she likes a place and wants to plot it on the map, she enters "x" in that place's row in column F to mark it. Otherwise she leaves the cell blank.

The user then uses Google's Static Map API, which, given a list of geo-coordinates, will return an image of a map with markers marking the locations. The user also wants to send a list of labels to be used on the markers. To add a marker given a pair of geo-coordinates and a label, the user needs to append

```
&markers=label:labelValue|latitude,longitude
```

at the end of the API. To do so, the user uses the following regular spreadsheet formula in cell G1:

```
=IF(ISBLANK(F1), "",
CONCATENATE("&markers=label:",F1, "|", D1, ",", E1))
```

where D1 and E1 store the latitude and longitude of the first item. What this formula does is to first see if F1 is blank. If it is, return nothing. Otherwise, return the concatenated string. She selects G1 and autofills other cells in column G.
Now the user gets all the "markers" strings she needs to compose the whole web API. She uses the `CONCATENATE` function again in H1 to combine the constant part of the URL with everything in column G using the formula `=CONCATENATE("http://maps.googleapis.com…", G:G)`. Now, H1 is the complete web URL that returns an image of a map with the makers specified. Gneiss provides an `IMAGE(url)` function (same as Google Spreadsheet's `IMAGE` function) that lets users display a image in a spreadsheet cell given a URL. So, the user enters `=IMAGE(H1)` in H2, and H2 becomes the actual map image. The result is shown in Figure 3.4.

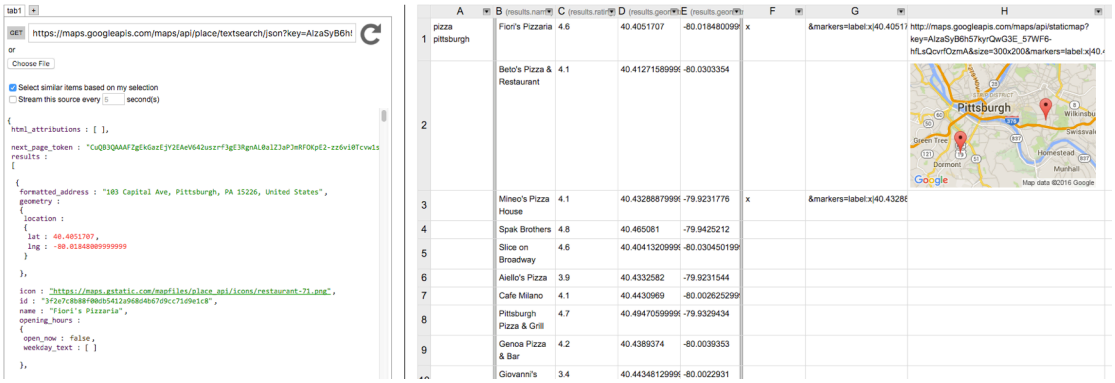


Figure 3.4. A screenshot of the demonstrative example in section 3.5.1

The created spreadsheet program is highly reusable. The user can search for another type of place by editing the cell A1, or add or remove a place from the map by entering or deleting contents in its corresponding cell in column F. None of these requires the user to change any programming logic in the spreadsheet. The user could easily share this city trip planner spreadsheet with her friends, with a little explanation about what the columns are and how to use them.

As mentioned in Chapter 1, Gneiss also allows users to turn data in the spreadsheet into a web application. I will demonstrate in Chapter 4 how the user can create a web application where the user can enter the search term in a textbox and interactively plot places on a clickable map using checkboxes (see section 4.2).

3.5.2 CUSTOMIZED BOOK LIST

Another advantage of using a spreadsheet is that it enables users to easily combine their own data with web service data, since spreadsheets are familiar tools for people to store personal data. Suppose the user keeps a list of book titles she owns in a spreadsheet and wants to get the publisher, year and reviews of the books [5]. Searching the book titles one by one on Google for reviews would be tedious. Instead, she could load the spreadsheet to Gneiss[6] and use Google's Book API to get the publisher and year of the books, and use the New York Time's Book Review API to quickly collect the reviews of the books. Google's Book API returns basic information such as the authors and publishing dates of a book. The New York Time's book review API lets users search the reviews of a book using the book's title, and returns a list of reviews where each review has a summary text and a URL to the full review on New York Time's website. Using Gneiss, the user could quickly retrieve the information for all her books by sending the first book's title as the query term, extracting the first book's information to the first row of an empty

[5] This use case is inspired by a real question on StackOverflow about getting book data using web services in Excel. See http://stackoverflow.com/questions/7601685/excel-vba-macro-to-query-amazon-web-services-for-book-data.

[6] Gneiss currently does not support loading an external spreadsheet file (such as an Excel spreadsheet). But one can imagine this feature to be easily added to Gneiss. However, Gneiss can currently load JSON files from the local disk.

column, and autofill that information for the rest of the books. Multiple reviews of a book will be put in nested cells to be in the same row with the book title to increase readability.

3.6 LIMITATIONS AND DISCUSSION

The current Gneiss prototype has several limitations.

3.6.1 SPREADSHEET USABILITY

As mentioned in section 3.2, to provide the necessary freedom to experiment with new ideas, I chose to implement my own spreadsheet editor in Gneiss, including all the spreadsheet functions along with the sorting, filtering and autofilling mechanisms. As a result, Gneiss lacks some usability features that are common in commercial spreadsheet tools. To name a few, first, Gneiss supports only a few spreadsheet functions (such as those described thoughout this dissertation and summarized in Appendix A) that I implemented for the sake of demonstration and for running the user study (described in section 5.5). In contrast, commercial spreadsheets such as Excel support hundreds of spreadsheet functions. Second, the autofilling gesture in Gneiss is not as intelligent as in commercial spreadsheets in inferring the new values based on example values. For example, in Excel the user can enter "Monday" and "Tuesday" in two cells, select them and drag down to fill in the rest of the days of the week. Excel's Flash Fill [36] can recognize patterns in example strings and use the patterns to create new values, such as extracting everyone's first name. Gneiss does not have those abilities. Third, Gneiss currently does not support undo and redo. Fourth, commercial spreadsheets provide many copying and pasting options, such as pasting multiple cells to match the destination formatting or pasting by computed values only. Gneiss currently only supports copying and pasting a single cell at a time and can only paste the cell's input value.

Adding the various missing usability features to Gneiss is a future work. In this dissertation, I focus on describing the research contributions of Gneiss and also demonstrating that conventional spreadsheet features can be intuitively added to Gneiss with very little modifications.

3.6.2 SUPPORTING MORE TYPES OF WEB SERVICES

There are two main limitations of the current Gneiss prototype in supporting web data services. First, Gneiss supports only RESTful web services that return JSON data. It does not support web services using other protocols, such as SOAP, or returning other kinds of data formats, such as XML or CSV. However, I believe that the key features described above should still be able to be applied to those kinds of web services with minor changes in the source pane to accommodate the

differences. For example, the same drag-and-drop gesture for extracting a field in a JSON document can also be used to extract a field in a XML document.

The second limitation is the usability of web data services. Gneiss lets users load data from web services using a URL bar. This is a simple design that enables users familiar with web services to easily use any basic REST web services they want. However, for users who are not familiar with web services, they might need more guidance than a URL bar to use a web service. For example, they may prefer to have several built-in web APIs and to configure the parameters through dialog boxes. The URL bar also fails to support more complicated web services that may require special widgets or dialog boxes to configure. For example, some web services limit the number of data returned in each call and use special parameters to retrieve the next set of data. Gneiss currently does not have a good way to support automatically retrieving the next set of data using the URL bar besides having the user manually configure the parameter and retrieve the data again. The URL bar also does not support web services that use additional authentication protocols, such as OAuth.

Obviously, hardwiring a web API and a custom configuration dialog box for people to use a web service in Gneiss can be done (and many prior research systems such as [49,91] let people use web services in this way), and most of the novel features described above could still be used to support interacting with that web service data once they are retrieved. However, I had previously developed a tool to let users add a new web service to use in a finished application as a plugin. This tool, called Spinel [17], consists a library for application developers to create such an extendable application, and a GUI tool for end users to create a data source plugin without having to write any code. Spinel was originally developed for Android applications, but it can be applied to web applications like Gneiss as well. If integrated with Gneiss, the Spinel architecture could allow web services that cannot be used in a simple URL bar to be added to Gneiss as plugins without having to edit Gneiss' source code. It could also let Gneiss provide GUI widgets to help people use a plugin web service. Next, I briefly describe Spinel. The full description is published elsewhere [17].

3.7 SPINEL: ADDING DATA SOURCES TO AN APPLICATION AS PLUGINS

Many applications that people use daily are applications that provide user interfaces to interact with some backend data sources, and many of those applications have similar user interfaces. For example, there are hundreds of applications that let people search data about different types of places (such as Google Places, Yelp, Hotels.com, etc.), and these applications all show the data in either a list or a map. Spinel is motivated by this observation that a single, consistent user interface application (e.g., the same map app) may be used to display many different kinds of data (e.g., Google Places, Yelp, and other location data sources). Therefore, it aims to

help developers create an extendable application where the backend data sources can be easily added and removed by end users to make the application more reusable and tailored to individual needs. Spinel contributes a software architecture that allows applications to take a data source as a plugin. Moreover, Spinel provides a GUI tool to let people create such a plugin without having to write any code, so end users may be able to make their own plugin for the data sources they want (see Figure 3.5).

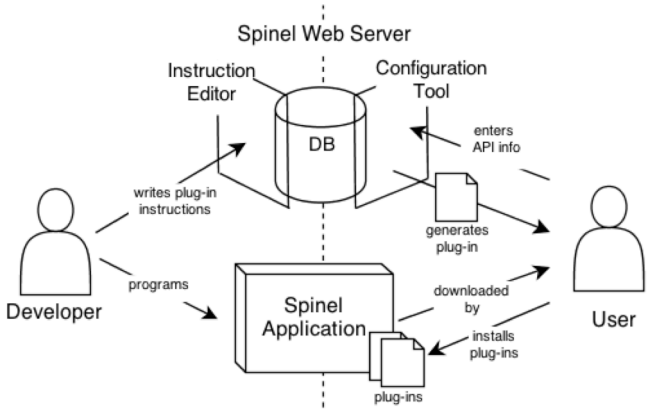


Figure 3.5. Spinel's architecture and modules

For example, I previously created a mobile text entry tool called Listpad [16] that can use relevant web services to provide autocomplete suggestions to help users enter data, such as using Yelp as the source for autocomplete suggestions when the user is entering a personal restaurant list. Listpad uses the Spinel architecture and is extendable because the user can add a new data source to Listpad to use as autocomplete suggestions by installing a plugin without having to edit Listpad's source code or rebuild the Listpad app.

One can imagine that Gneiss could also use the Spinel architecture to provide web services in the source pane. The same source pane UI, along with a dialog box mechanism for configuring parameters in a web URL, can be used on different APIs from different web services. The mappings between the Gneiss's UI and the web API used would be described in the data source plugin file. The plugin file could also include advanced authentication information, such as parameters required for OAuth (which is the predominant authentication protocol for modern REST web services), to allow more types of web services to be used in Gneiss. In this way, new web services could be easily added to Gneiss as plugins. These plugins could be made using the Spinel GUI tool and would then be independent from the main Gneiss application.

For example, if a user wants to use in Gneiss his own company web service that uses OAuth, and he knows all the required parameters, he could create a plugin for his company web service using Spinel's GUI tool, and add this web service to Gneiss by installing a plugin using a menu command in Gneiss. The web service could then appear in Gneiss's source pane as one of the available data sources. As the created plugin is an independent file, the user could also share the plugin with other people,

such as his colleagues, so they can also use it in Gneiss without having to create a plugin of the same data source themselves.

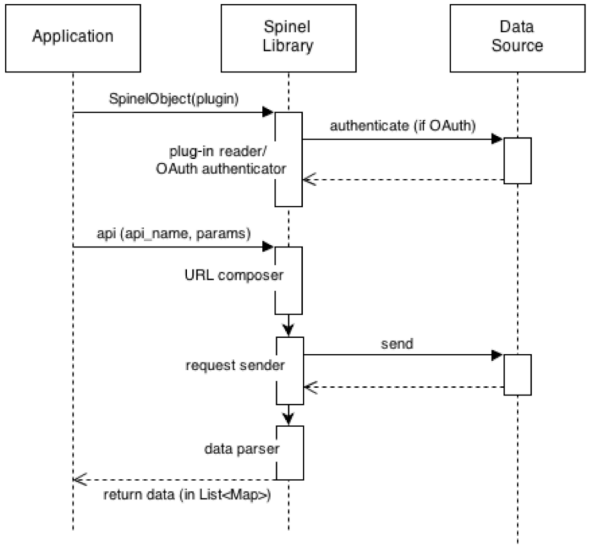


Figure 3.6. A UML sequence diagram showing a use of the Spinel Library. Time goes down from the top.

Continuing with the description of the Spinel architecture, it consists of a developer side to help application developers program an extendable application (such as to help make Listpad and Gneiss extendable), and an end user side for creating a data source plugin (such as to help the user in the previous paragraph to create a new plugin to add a new data source to Gneiss). Figure 3. shows how the developer side and the user side work together. On the developer side, Spinel provides a library that helps developers create and test an application (called a "Spinel application" in Figure 3.5.) without having to specify a fixed data source. The use of the Spinel library is shown as a diagram in Figure 3.. Spinel currently supports REST JSON web services and OAuth authentication. In the application, a web service is described using a `SpinelObject`. A `SpinelObject` can be initialized manually by the developer or by reading a plugin file created by the users, which is a file in JSON format describing the information of the web service. When the application starts, it goes to a predefined directory that has all the plugin files to read all the plugins and create the `SpinelObject`. The library further provides many convenient functions to use a `SpinelObject` for authentication and exchanging data web services.

To enable people who do not know about the application's source code to create a data source plugin to use in the application, the developer has to specify what types of data this application needs. For example, suppose in Gneiss there would be a widget to let users retrieve the next set of data using a web API. To do this, the widget needs to know what parameter in the API controls the "page number" of the data. Spinel provides a web-based plug-in instruction editor for the developer to describe this. So continuing the previous example, to let users create a plugin to use in Gneiss's widget, the developer would specify that the user needs to mark the parameter for retrieving the next set of data when she makes a plugin using Spinel's *plugin configuration tool*, which I will explain next (how to mark a parameter in a web API is shown in Figure 3. at 2, where the user marks the parameter "q" in a web API to be where the query term should go so the application can know how to use this API).

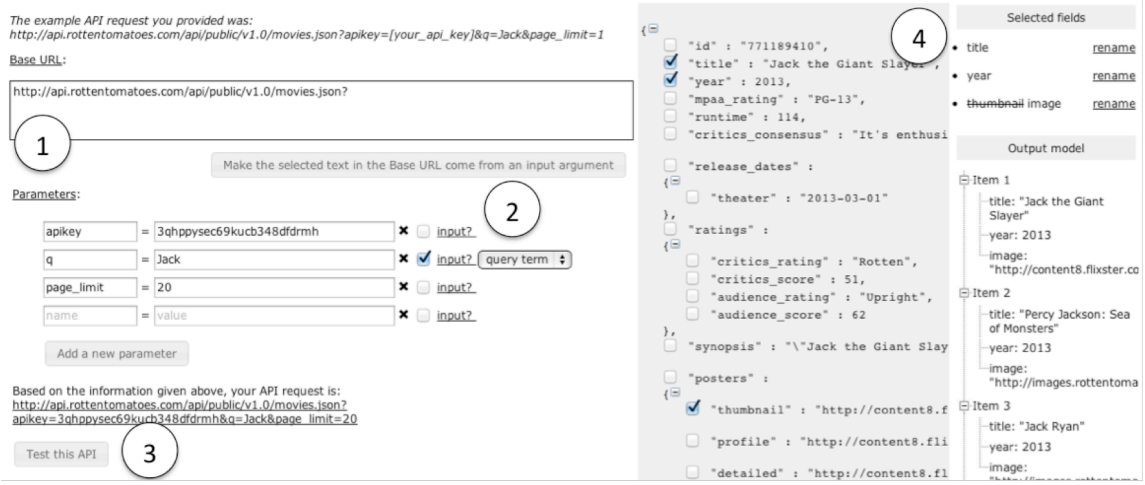


Figure 3.7. The main API editing page in the Spinel web-based plug-in configuration tool for users. (1) An API is divided into a base URL and a set of parameters, which are filled in by the tool if the user provided an example API request on a previous page (not shown). (2) The user selects a parameter to be an input field and gives it an appropriate name using the drop-down menu that is generated based on the developer's instructions. (3) The user can click on the "Test this API" button to see the return data (4, with the gray background). The user can select the desired fields as the final output of this API call using the checkboxes. The tool shows all selected field names and visualizes the output model that will be sent to the application in a tree at the far right, to help the user confirm that her selections are correct.

To create a plugin for an application to use a new web service, Spinel provides a web-based plugin configuration tool (Figure 3.) where the user can do so in a graphical interface without writing any code. The configuration tool uses the developer's instructions to guide users in creating and testing a data source plugin, such as adding the required APIs (Figure 3. at 1), marking the required parameters in an API (Figure 3. at 2) and the required fields in the return document (Figure 3. at 4). Similar to Gneiss, Spinel's configuration tool also uses a "programming-with-example" style to lets users start creating a plugin by giving an example API (since many web services provide example web APIs on the website to teach people how to use them) and then parses it to a more readable and editable format (Figure 3. at 1. Gneiss could also provide a dialog box similar to this to help users edit a web API). The user can download the created plugin and add it to an application to use the web service, and choose to share the plugin on Spinel's server for other people to use.

More details of Spinel are described elsewhere [17]. As future work, Gneiss could be adapted to use the Spinel architecture to provide easier-to-use interfaces to several built-in web services and allow additional web services to be installed as plugins to help people who are not familiar with web services.

3.8 CONCLUSIONS

In this chapter, I presented a spreadsheet model for using web data services. Gneiss extends the familiar spreadsheet language and drag-and-drop interaction to support constructing two-way data flows between multiple web services and a spreadsheet editor. Users can extract structured web service data by example without having to write any code, and Gneiss extends the spreadsheet language syntax to let structured data be used in spreadsheet formulas. Spreadsheet sorting and filtering are executed dynamically as a mechanism to further refine newly retrieved data. The familiar autofill gesture is extended to support sending a batch of similar web service requests. Gneiss abstracts away the complexity of dealing with different states of an asynchronous network call using the spreadsheet's live constraint evaluation model to automatically send requests, update data and computations, and handle failed calls for the users. Moreover, Gneiss can generate parallel-running data extraction programs using the dependency among spreadsheet cells.

With Gneiss, the user can create a variety of spreadsheet programs that make custom use of multiple web data services as I demonstrated in the examples in this chapter. However, a spreadsheet is still less usable and sharable compared to a web application. For instance, in the first demonstrative example, the created spreadsheet that lets users search places and plot them on a map by editing spreadsheet cells may be quite difficult to use and read on a mobile device compared to a web application that shows data in a grid list and provides more GUI controls to handle user inputs. In the next chapter, I will present how Gneiss enables users to program interactive web applications that make use of the data and computational logics in a spreadsheet.

CHAPTER 4 PROGRAMING DATA-DRIVEN WEB APPLICATIONS[7]

Many websites that people use daily are *web data applications* – applications that use backend data, supporting the searching, sorting, filtering and visualizing of the data based on the user input. For example, Yelp lets users search for restaurants and apply filters to show only the ones that accept credit cards. Expedia lets users search for hotels, sort the results by price, and plot the results on a map. Creating this kind of application requires a person to program a responsive web interface that can dynamically manipulate backend data and create a display based on the results. This usually involves writing complex code that has multiple nested callbacks to handle user events and perform the appropriate actions such as firing web service requests, retrieving new data, and repopulating the interface.

4.1 MOTIVATION, CHALLENGES AND CONTRIBUTIONS

In the previous chapter, I had discussed the programming challenges of using web data services and how a spreadsheet model could be extended to address those challenges and enable users to create spreadsheet programs that exchange data with multiple web services without writing conventional code. However, new challenges arrive when trying to create a *web application* that uses backend data sources.

One challenge is to program web pages that hold dynamic data instead of static content. In basic web programming, a web page's layout and content are written in an HTML file. Many end-user tools (such as Adobe Dreamweaver) provide WYSIWIG HTML editors to let people edit the content and view the rendered page at the same time. However, for web pages that use backend data sources, the content is often dynamic, retrieved from a remote database based on how the user interacts with the web page. The dynamic content is injected into the HTML page using a separate JavaScript or PHP file. Using a simple news web application for example, the first page often shows headline news that is retrieved when the user loads the page in the browser. When the user clicks on a news link, the content in the next page is dynamically generated based on what news item the user had clicked. Dealing with dynamic content significantly increases the programming complexity, as a person now needs to write more advanced code in languages other than HTML and CSS to link the web page and the data. It also becomes difficult to use a WYSIWYG editor to edit and view a web page since many GUI elements in the page are placeholders that have no content.

[7] The research in this chapter was also described in our publication at UIST'14 [19]

Another challenge is to program data-related interactive behaviors in the web application. Most data-driven web applications let users search, sort, filter and visualize data interactively using GUI controls. The user's actions in the web page sometimes also trigger new queries to be sent to remote data sources, such as entering a new search term in a textbox. Currently, programming interactive behaviors in web applications is mostly done using JavaScript event handlers to capture user input events and execute the corresponding actions such as modifying some data or changing the user interface. For users who are not familiar with JavaScript programming, this adds another level of difficulty to creating a useful data-driven application.

To address these challenges, my dissertation contributes a spreadsheet model that supports programming interactive, data-driven web applications. Gneiss's programming environment integrates a WYSIWYG web interface builder for creating web pages. The new spreadsheet model unifies the access to web GUI elements. All GUI properties have the capabilities of spreadsheet cells and can be referenced anywhere in the spreadsheet using a spreadsheet-like syntax. Similarly, the value of any GUI element property can be a constant or a value computed using a spreadsheet formula that depends on other cells. This enables users to use spreadsheet languages to program data bindings between the web application and the spreadsheet editor, which can hold constant data entered by the user and dynamic data retrieved from web data sources as described previously. The data flow between the web interface and the spreadsheet editor is two-way, allowing the user's interactions with the web application to modify data in spreadsheet cells and further trigger different actions at run time such as to fire web service calls, retrieve new data, apply new sorting and filtering rules, and subsequently repopulate the web interface. Gneiss' spreadsheet model not only provides a live environment for programming and testing data-driven applications, but also enables a "programming-with-example" style [64] that allows users to create a web interfaces with visible example data that were retrieved from actual data sources instead of working with placeholders and invisible data as required when programming in HTML and JavaScript.

Also, the new spreadsheet model facilitates two-way communication by providing the "once-around" semantics of one-way constraint solvers, which has been shown to be useful in previous systems [62,63,70]. This, along with the dynamic properties of GUI elements that are triggered when the user interacts with the elements, and new formulas that control when a spreadsheet cell is re-evaluated, enable the user to program a wide range of interactive behaviors solely using a "pull model" that is consistent with the current spreadsheet formula evaluation paradigm. It eliminates the need for a conventional event-based "push model" that is used by other GUI tools and toolkits.

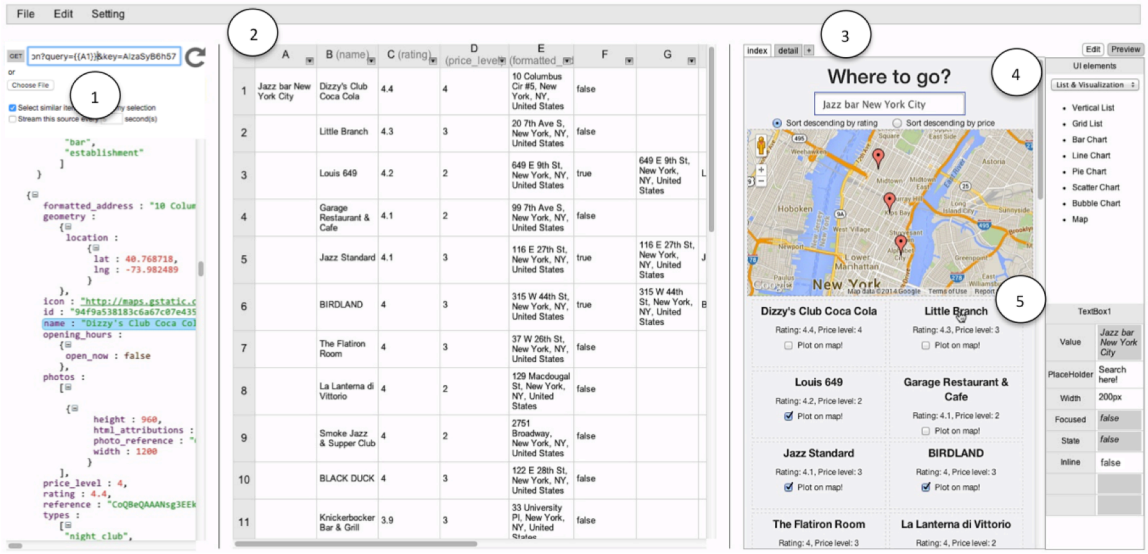


Figure 4.1. A screenshot of Gneiss showing the web application created in this chapter's usage scenario. (1) is the source pane where the user can load a web API in the URL box and extract the desired fields from the return data to the spreadsheet editor through drag-and-drop. (2) is the spreadsheet editor that stores and manipulates the data to be used in the web application. (3) is the web interface builder where the user can create a web application by dragging-and-dropping GUI elements from the toolbar on the right (4) to the output page. The user can select a GUI element in the output page (the selected element is highlighted with a dark blue border, which currently is the textbox at the top of the page) and view its properties in (5). Property values are cells that can contain formulas and can be referenced by other cells.

4.2 USAGE SCENARIO

Here we describe a scenario where Ted uses Gneiss to create a web application that helps him decide where to visit in a city. The application has a textbox that searches an online place database and shows the search results in a grid list. The user can choose to sort the results by rating or price using two radio buttons. Each item in the grid list contains the name, rating, price level of a place, and a checkbox that lets the user display the place on a map. Finally, the user can click on a place name to go to a "details" page that shows photos of the place retrieved from an online photo database. Figure 1 is a screenshot of part of the end result (with the created web application in the right pane). We use this scenario to give an overview of the features described in this chapter.

Similar to the background of the user in the usage scenario described in the previous chapter (Section 3.3), Ted is a fluent spreadsheet user familiar with sorting, filtering and using functions in spreadsheets, and knows what web APIs he should use for his tasks. Ted also has some prior experience about using a web interface builder to create web pages.

Ted first enters Google's Place Text Search API in the URL box in the source pane (Figure 4.1 at 1). The returned data, in JSON format, are shown below, and Ted can use this example data to demonstrate the next steps. After seeing that the API works

with a constant string, Ted wants the value of the "query" parameter in the web API to be whatever the user types in the text box in the web application at runtime. To make the query string be dynamic, Ted first sets the value of the parameter to be the value of cell A1 in the spreadsheet, using the syntax `{{A1}}` (Figure 4.1 at 1, in the URL box), as discussed at length in chapter 3.4.2. He then extracts the data he wants from the query result – the name, rating, price and address fields – to the spreadsheet by dragging-and-dropping each value into its own column (Figure 4.1 at 2, columns B-E).

Ted now starts to create the web application using Gneiss's right pane, which is a web interface builder. He first drags a text label (for the header) and a text box (for entering the query string) from the tool bar at the right of the interface builder (Figure 4.1 at 4) to the output page. In the web interface builder, Ted can select any GUI element in the output page (which will then be highlighted with a blue border), and view and edit its properties, which are shown as a small one-column spreadsheet at the bottom right (Figure 4.1 at 5). Ted selects the heading text and changes its value to the constant "Places to Go". Ted then selects the text box. The "value" property of the text box is a dynamic property depending on what the user enters at runtime (currently "Jazz bar New York City"). Back in the spreadsheet, Ted enters `=TextBox1!Value` in cell A1 to make it use the text box's value.

Now every time Ted types in the text box in the web interface builder and hits enter, cell A1 in the spreadsheet changes to the text box's value. As described in chapter 3, this will trigger the system to fire a new API request using A1's new value (which is the text box's value) as the query term (by default, a textbox's value property changes when the user presses the enter key. The user can change the "live" property of a textbox to be "true" to let its value change as soon as the user types each character into the textbox, but that is not appropriate for this scenario as incomplete strings would be sent as queries to the web service). This, in turn, will refresh the extracted data in spreadsheet columns B-E with the latest retrieved data. To display the search result in the web application, Ted drags a grid list to the output page. In the first item of the grid list, Ted drags two text labels, and the system populates the rest of the grid items with the same UI objects. Ted wants the top text item in each grid item to be the name of the place. To do so, Ted selects the top text in the first grid item, and changes its value property to be `=B1`. The top text in each of the rest of the grid items automatically changes to be the value of the corresponding cell in column B. Ted then makes the second text label in each grid item show the rating and the price level of a place by setting the value property of the second text label in first item to be `= "rating: "&C1&", price level: "&D1`, and letting the system populate the rest of the items. Ted makes a few tests by entering new values in the search textbox in the web interface. He can see that the grid list in the web page updates automatically to show the latest search results. The length of the grid list changes according to the number of search result items returned.

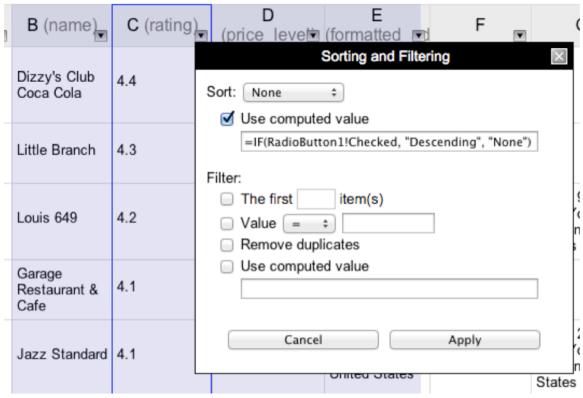


Figure 4.2. Sorting and filtering rules can be computed from GUI element properties in a web application using a spreadsheet formula.

Ted then drags in two radio buttons to the web page to control how the search results are sorted. Ted changes the label of the first radio button to "Sort descending by rating". Then he clicks on the small arrow icon at the top of column C, the column that stores the ratings, to open up a dialog box that controls the sorting and filtering of that column. Ted sets the sorting rule to come from a computed value, and enters the formula `=IF(RadioButton1!Checked, "Descending", "None")` to sort the results descending by the rating column if the first radio button is checked (Figure 4.2). Ted uses the same method to set up the second radio button to control if the results are sorted by price (column D). Now as Ted toggles the radio buttons in the web page, he can see the spreadsheet data being sorted accordingly, and the grid list in the web application being updated at the same time to reflect the results.

Ted's next task is to let people be able to view selected places from the search results on a map. To do so, Ted drags a checkbox to the gird items and sets its label to be "Plot on map!". In the spreadsheet, Ted enters `=CheckBox1!Checked` in cell F1 to get if the checkbox in the first grid item is checked. Ted then selects F1 and uses the standard spreadsheet mechanism to drag down to fill in column F with the values of the "checked" property of each of the checkboxes. Now as Ted checks and unchecks a checkbox in the web interface, its corresponding cell in the spreadsheet column F changes between "true" and "false" too. In cell G1, Ted types in `=IF(F1, E1, "")` and autofills the rest of column G. This formula fills in the cell in column G with the address cell (column E) in the same row if the checkbox cell in that row (column F) is checked. Finally, in the web interface pane, Ted drags a map visualization from the toolbar to the output page, and sets the "addresses" parameter of the map to be all cells in column G by typing in `=G:G` (the standard spreadsheet syntax to reference a whole column). Now Ted has an interactive map that dynamically displays the checked places.

The last step to finish this web application is to allow the place's name to be a hyperlink that goes to a new page that shows photos of the place from an online place database. This demonstrates how people can use Gneiss to create a multi-page application where the content of the next page is generated dynamically based on what the user selects in the previous page. This also demonstrates how people can use Gneiss to combine data from multiple data sources.

To do so, Ted creates a new page called “Details” by pressing the “+” icon next to the page tabs at the web interface builder (Figure 4.1 at 3) and enters the name of the new page in a dialog box. He then selects the place name text, and uses the property sheet at the lower right to set its “link” property to be “Details”. This causes a place name text to be hyperlink so that when clicked, it will bring the Details page to the front. Ted then needs to get the latest clicked place name in order to send a different API request to retrieve the photos for that place, and fill in the details page. In Gneiss, all UI objects have a “state” property that reflects how the mouse cursor interacts with it. For example, when the mouse moves over the text object and then clicks on the text, the state for the text changes from “idle” to “hovered” to momentarily be “clicked” then goes back to “idle”. Ted autofills column I with the state property of the corresponding place name text. Then in cell J1 he types:

```
=IF(COUNTIF(I:I, "clicked")>0,LOOKUP("clicked", I:I, B:B),J1)
```

This uses the standard spreadsheet functions `COUNTIF` and `LOOKUP`, and will count how many cells in column I have value “clicked”. If there are any, then cell J1 becomes the cell in column B (the name column) that is in the same row with the “clicked” cell in column I. Otherwise the formula does not change J1’s value. This makes J1 be the name of the most recently clicked place. Ted then uses Flickr’s Photo Search API and cell J1 as the query string to search for the photos of the clicked place, and uses similar methods as before to extract the photo data returned by Flickr and show the photos using a grid list in the web interface builder. Lastly, Ted adds a “Back” button to the Details page to let the user go back to the index page. Note that values are passed around and interactions are triggered using standard spreadsheet functions and formulas, without the need for any event-based programming.

Ted now has finished creating the application! He plays with the application in preview mode, does a few test queries, and exports the application when done. Gneiss generates a URL that can be used to open this application anywhere in a browser. Ted sends this URL to his friends to share the application.

4.3 KEY FEATURES FOR CREATING INTERACTIVE WEB APPLICATIONS

In this section I describe the key features in Gneiss to assist people in creating interactive, data-driven web applications.

4.3.1 CREATING A DYNAMIC UI THAT SHOWS SPREADSHEET DATA

As in many commercial web interface builders, Gneiss lets users create UI elements in a web page by drag-and-drop. As mentioned in chapter 3.2, I implemented my own web interface builder, and thus Gneiss currently supports a limited set of UI

elements. Gneiss currently supports text and image objects, input elements (such as text boxes and buttons), lists, and visualizations, with the full set of HTML5 elements, properties and GUI controls as obvious future work. In general, the properties of these elements include string properties (e.g., the text in a heading, the label of a button), styling properties (e.g., color and width), link properties, and interactive properties that change values as the user interacts with the element (explained in detail later). String and styling properties let users style GUI elements using conventional mechanisms. A Link property turns a GUI element into a hyperlink that goes to the value of the property. The value can be a URL or the name of another page opened in the interface builder, allowing the user to create a multi-page application. The user can specify how the application transfers from one page to another, and how the contents of one page may depend on values from another, as described in the usage scenario. A complete list of currently supported web UI elements and their properties is in Appendix B and discussions of extending this list is in section 8.1.5.

In Gneiss, the user can edit a GUI element property just like editing a cell in the spreadsheet editor. A GUI element property can be a constant or a dynamic value computed from the spreadsheet data. Setting a property value to use spreadsheet cells will cause the property be recomputed every time that these cells change. For example, if the user sets the "value" property of a text object to be `=CONCATENATE("Rating: ", C1))`, then each time that cell C1 in the spreadsheet editor changes, the text object will also change to show the latest value.

4.3.1.1 Dynamic Lists

Gneiss currently supports vertical lists and grid lists. Lists have a "Populate" property that if true will fill in all items of the list with the corresponding elements based on edits to the first item. For example, in the usage scenario, Ted only needs to drag two text labels and a checkbox into the first item of the grid list, and the rest of the items in the list are populated with the same objects as are in the first item. Property values of a UI element in a list also populate in the same way, so changing the property of a UI element in the first item of the list will affect all of the corresponding UI elements in the rest of the items of the list. If the property value is a constant, all other UI elements are populated with the same value. If the property value is a formula that computes the value based on some cells in the spreadsheet, the system will populate other UI elements in the list with the corresponding cells in the same column but using the row based in the element's index. For example, in the usage scenario, Ted sets the value of the top text label in the first item of the grid list to be `=B1` (the name of the first place). The system automatically populates the top text label in the second list item to be `=B2`, and so on.

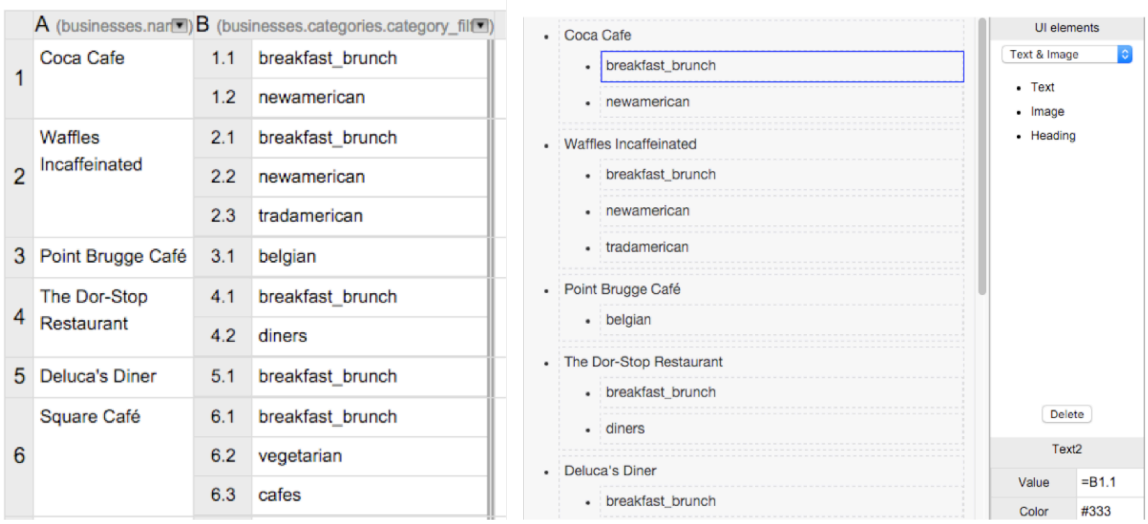


Figure 4.3. The user can display hierarchical data in the spreadsheet (left) as nested lists in the web application (right). Here, the user creates a first-level list to show restaurant names (column A), and within each restaurant in the first-level list the user drags a second list that displays the categories of the restaurant (column B).

Items in a list can be shown and hidden dynamically at run time based on the spreadsheet data. When the "Populate" property is set to "true", the system will adjust the length of the list to show the non-empty rows in the column that the UI element's value depends on. For example, in the usage scenario, the length of the grid list changes dynamically based on the number of search result items returned. The user can use the "MaxNumberOfItems" property to set the maximum number of items in a list object. The system will hide the empty items automatically.

Alternatively, the user can set the "Populate" property of a list object to "false". In this case, the list object becomes a pure layout object and the user can manually put different UI elements in different list items.

4.3.1.2 Nested Lists

As described in the chapter 3.4.5, Gneiss is able to visualize the hierarchies in data using nested spreadsheet cells. In the web interface builder, the user can also create nested lists to display nested data in the spreadsheet. For example, Figure 4.3 at the left shows a list of restaurant names in column A, and each restaurant has a list of categories displayed in nested tables in column B. The user can create a nested list such as in Figure 4.3 at the right to show the data in a web application. To create a nested list, the user drags a list object inside the first item of another list object, and lets the system populate the rest of the items. Spreadsheet cells used in a nested list are iterated over the corresponding nested row index. For example, in Figure 4.3 at right, the restaurant names are shown in the first level list by setting the first text in the first item in the first level list to be `=A1`. Each restaurant's categories are displayed in the second level list by setting the first text in the first item in the

second level list to be `=B1.1` (Figure 4.3 at the lower right corner). As the system populates the *second* restaurant's data, it first iterates over the first row index which therefore shows data in A2 as the restaurant name in the first-level list. Then the system goes to the second-level list and iterates over the second row index (B2.1 to B2.3) to display values in all nested cells in B2. Currently, how the nested indexes are iterated in a list is decided by the system. Future work can be to generalize this to let users be able to customize the rules. However, currently the users have complete control over the data in the nested table – they can click on a nested cell to edit it just like editing a regular cell. Users can also create a new column that has the same nested structure as a selected column, use formulas to compute values for that column using other cells in the spreadsheet, and show that computed column in the web application. For example, in Figure 4.3, the user might want to write a formula to make the text prettier for the categories by using string manipulation functions such as replacing underscores with spaces. Features for calculating new nested values are described in detail in chapter 5.3.2.

4.3.2 MODIFYING SPREADSHEET DATA FROM WEB APPLICATIONS

In Gneiss, user inputs into the web application can also affect the spreadsheet data, making the data flow two-way. As described before, GUI element properties are treated as spreadsheet cells in Gneiss. The user can reference any property of a UI element using the syntax `ElementID!PropertyName`, using a syntax similar to how a cell in another worksheet is referenced in conventional spreadsheets. References like this can be used not only in the spreadsheet editor but also in the GUI property sheets in the web interface builder as well. For example, the user can set `Text2`'s value property to be `=Text1!Value`, making the two text objects display the same content. We further added a convenient keyword `THIS` to let a GUI element property reference other properties in the same element using `THIS!PropertyName`.

Currently, a web GUI element's ID is assigned by the system and is not changeable. This is a limitation of the current prototype system. Future work includes letting users edit an element's ID in the same way as editing its other properties. For example, in the usage scenario, a user might want to change the search textbox's ID from `TextBox1` to `SearchBox` to make the ID more meaningful.

4.3.2.1 "Autofilling" GUI Property Values

We extend the autofill gesture to facilitate referencing properties of populated GUI elements when they are in a list object. The user only needs to enter a reference into a spreadsheet cell for the desired property of the UI element in the first list item, and then select that cell and autofill down. For example, in the usage scenario, Ted types `Checkbox1!Checked` in cell F1, selects F1 and autofills down to F10. The system then fills in F2 to F10 with the "Checked" property of the checkboxes in the

second to tenth items in the grid list. The IDs for the checkboxes in lists are automatically assigned as *`FirstElementID-Index`*. For example, in the usage scenario, the ID of the checkbox in the second list item is `Checkbox1-2`. Therefore when the user autofills the checkboxes' checked properties in column F, the system inserts `=Checkbox1-2!Checked` in cell F2. The system does not make this naming convention transparent to the user, as the user is not able to select and edit a populated GUI element from the web interface builder (since all its properties are maintained by the system).

4.3.2.2 Using Web GUI Controls to Sort and Filter Data

As described in chapter 3.4.3, Gneiss supports two levels of dynamic sorting and filtering of web service data. Not only are the sorting and filtering rules reapplied every time when new data are retrieved, but also can the rules themselves be computed dynamically using formulas. Since Gneiss unifies the access to web GUI elements in the spreadsheet model to treat GUI element properties as spreadsheet cells, sorting and filtering rules can even be computed based on GUI element properties in a web application. This allows the user to program the interactive behavior to be based on the values of various GUI controls such as checkboxes, radio buttons or sliders to sort and filter web service data shown in the web application.

As described in chapter 3.4.3 and in the usage scenario in section 4.2, to specify computed sorting and filtering rules, the user would open the dialog box of the column to sort and filter the data, check the "Use computed value" checkbox and in the textbox below enter a spreadsheet statement that returns a string of predefined values that represent the rules (listed in **Error! Reference source not found.**). So for example, to sort the column only when `Checkbox1` is checked, the user enters `=IF(Checkbox1 !Checked, "Descending", "None")` as the rule in the "Use computed value" textbox for sorting. Or if the user wants to program the behavior of using a slider to filter the data, she can enter a formula like `="Filter value, >=, "&Slider1!Value` in the "Use computed value" textbox for filtering to filter out cells that are less than the value of `Slider1` in the web application.

4.3.3 MAKING APPLICATIONS BE INTERACTIVE

One of the innovations in Gneiss is the way that users can make their web applications be interactive. Originally, we explored having Gneiss use a conventional event-based or callback architecture like Java and JavaScript, where UI elements would contain actions to be performed when operated. However, it is awkward to combine these "push" actions (where the action routine in a UI element would set other cells—pushing values to them) with the spreadsheet "pull" model (where cells compute their own values with formulas by pulling in the needed values). Therefore, I designed a way for the Gneiss user to define interactive behaviors without ever

needing to write any callback or event procedures. As Gneiss is a live and visual programming environment, I wanted the user's interaction to be live, with all of the values being visible in Gneiss in the same way that spreadsheet cells are normally.

To achieve this, I designed a new way to connect the spreadsheet to properties of GUI elements. The interactive properties of a GUI element show the value set by the user. These properties are displayed in the property sheet in the web interface builder using grey color cells to show that their values are not editable there. Instead, interactive properties change values live based on how the user interacts with the GUI element. For example, all GUI elements have an interactive property called "`State`" that shows how the mouse cursor interacts with them. The "`State`" property changes its value between "idle", "hovered", "pressed" and "clicked". For events that have a very short duration, such as "clicked", the value will stay a few extra milliseconds after the event so the user can notice that it happened and use that value in spreadsheet formulas. Other interactive properties are mostly for data input elements, such as "`Value`" for text boxes and sliders, and "`Checked`" for radio buttons and checkboxes. The user can easily test the interactive properties by interacting with different GUI elements in the web interface builder and viewing the changes.

Interactive properties can be used in the spreadsheet cells to compute different data based on the user's action. In both the spreadsheet editor and the property sheet, values of interactive properties change dynamically in keeping with Gneiss's "programming-with-example" style. One limitation of the design of interactive properties, though, is that the user cannot programmatically set the value of an interactive property, such as to set the initial value of a textbox, as the value is always decided at run time based on how people interact with the application. I discuss this further in the limitations section of this chapter in section 4.5.

4.3.3.1 Constraint Evaluation and Circular Constraints

Like some other one-way constraint solvers [62,63,70], our spreadsheet formula solver provides "once-around" semantics for circular constraints. Here I use an example to explain how this works. Suppose the user sets A1 to be 0 and A2 to be `A1+1`, which makes A2 be 1. The user then sets A1 to be `A2-1`, which creates a circular reference. Most conventional spreadsheets such as Excel do not allow circular references and thus will return errors in both A1 and A2. In contrast, Gneiss allows circular references, and always evaluates any dependent cells exactly once when any cell in the cycle changes. Here, the system starts computing A1 by asking A2's value. Since A2 is `A1+1`, it goes back to A1 and finds that it has reached the beginning of the cycle. The system then stops the circular reference here and returns A1's old computed value, 0. That makes A2's value be 1 (0+1), and A1's value stays 0 (1-1).

Supporting “once-around” circular constraints in the spreadsheet makes more types of expressions possible. For example, a spreadsheet cell can now reference itself. The system will return the cell’s original value before it is recomputed. Combined with the `IF` statement, this allows the user to set a cell’s value to something if a condition is true, otherwise having the cell *retain its original value*. This is a common behavior many modern programming languages where a programmer specifies an `if` condition without having a `else` condition. In Gneiss, the user can program the same thing using spreadsheet languages by having the “else” parameter in the `IF` function be a reference to the cell itself, or leave the else condition blank. For example, in the usage scenario, if the user adds a “search” button next to the text box and wants the system to send the search term only when the “search” button is clicked, she can enters `=IF(Button1!State=”clicked”, TextBox1!Value, A1)` (or `=IF(Button1 !State=”clicked”, TextBox1!Value)`, omitting the else condition) in cell A1. Now when `Button1` (the ID for the search button) is clicked, its `State` property will become “clicked” for a few milliseconds, making the condition in the `IF` statement be true and thus changing A1 to be `TextBox1`’s value. When `Button1`’s `State` goes back to “idle” or other values that are not “clicked”, the `IF` statement goes to the “false” condition and references A1 itself, thus A1 remains the same value. This creates the desired behavior the user wants.

This expression is useful to program interactive behaviors as it allows the user to set a cell’s value when a GUI element enters a certain state, but does not change the value back when the element enters another state. For example, suppose the user wants to change a blue hyperlink text to be red once it is pressed. This can be done simply by entering the formula `=IF(THIS!State=”pressed”, “red”)` in a text element’s color property. Now despite the “State” of the text will go back to “idle” from “pressed” after the user releases the mouse button, using a self-reference, the text is able to stay red.

4.3.3.2 Timers, Animations and Refreshing Constraints

While Gneiss is not designed for creating animation-heavy applications such as games or art websites, it does provide a few functions to program simple animations. Gneiss has a `TIMER(exp, ms)` function that will run the expression `exp` every `ms` milliseconds. For example, entering `=TIMER(IF(THIS!State=”hovered”, THIS!Width+10), 15)` as a GUI element’s width property value will increase its width by 10 pixel every 15 milliseconds when the element is hovered by the mouse cursor. I also designed a convenient function `ANIMATE(startValue, endValue, ms)` that animates `startValue` to `endValue` in the duration of `ms` milliseconds. The function uses jQuery’s animate function. `startValue` and `endValue` have to be numerical values. Another example is presented in the demonstrative examples section 4.4.2.

As in most conventional spreadsheets, by default when a spreadsheet cell's value changes, the cell will invalidate all other cells that directly depend on it. This causes the dependent cells to re-compute their values. If the value of that cell changes, then the cell continues to invalidate other cells that directly depend on it, and so on. However, if the value stays the same, then the cell will *not* invalidate its dependent cells, and the propagation stops. This increases the system's performance and also ensures that web API requests are not run when a parameter value is updated to the same value as before. This tends to happen with formulas that contain `IF`s or other control structures, where the constraint must be re-evaluated, but ends up calculating the same value.

Gneiss also offers a function `REFRESH(exp)` to let users invalidate all children of a cell every time the cell is recomputed (even if after recomputing, the cells value does not change). For instance, if cell A1 is used as the query parameter of a web service call, setting A1 to `=TIMER(REFRESH("world cup"), 600000)` will invalidate the children of A1 every 10 minutes even though A1's value is always "world cup", triggering the system to query the web service every 10 minutes to retrieve the latest search results for "world cup". I described in the previous chapter that the user could force re-evaluating *all* formulas in the spreadsheet using a menu setting. This `REFRESH` function acts differently as it only re-evaluates the direct dependencies of a cell.

4.4 DEMONSTRATIVE EXAMPLES

Here we use three more examples to demonstrate our system's ability to build elaborate interactive interfaces and support different types of web applications.

4.4.1 PHOTO SLIDESHOW

Slideshow is a common UI design to show a list of photos. It usually contains a large photo at the center of the interface, "previous" and "next" buttons that let the user change the photo displayed by going up and down the list, and thumbnails of all photos at the bottom of the

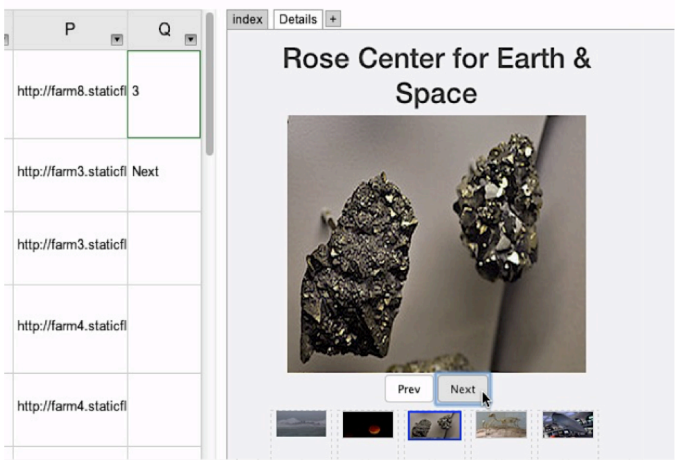


Figure 4.4. A screenshot of making a slideshow interface with photos from Flickr. At the left, column P stores a list of photo URLs, Q1 stores the current slideshow index (3), and Q2 stores the button that the user last clicked on ("Next"). At the right is the slideshow application. The thumbnails show the photos in column P. The large image is the photo at row 3 (the index value stored in Q1) and the 3rd thumbnail is highlighted.

interface. I will show here how the user can create a photo slideshow interface using Gneiss and spreadsheet functions. The screenshot of the created application is in Figure 4.4.

First, the user would retrieve a list of URLs of photos using a photo web service (such as Flickr) and stores the photo URLs in column B, in the same way as described in the usage scenario. The user then filters the search results to see the first 5 photos. In the web interface builder, the user can drag an image object to the output page and two buttons before and after the image, changing the label of the buttons to be "Previous" and "Next". The previous and next buttons control the index of the photo being shown. The user decides to store the index of the current photo in cell A1. She types in "0" in A1 first. She then needs know if the "Previous" button or the "Next" button gets clicked. To do so, she types in A2:

```
=IF(Button1!State="clicked", "Previous",
IF(Button2!State="clicked", "Next", A2))
```

where `Button1`is the previous button and `Button2`the next button. This nested `IF` formula changes A2 to be set to either "Previous" or "Next" when a button is clicked.

Now the user needs to add or subtract the index cell (A1) by one based on what button is clicked. She changes A1 to be:

```
=IF(AND(A2="Previous", A1>1), A1-1,
IF(AND(A2="Next", A1<5), A1+1, A1))
```

This nested `IF` formula not only updates A1's value based on what button is clicked, but also makes sure it stays between 1-5, as the photo URLs are in rows 1 to 5 of column B. (To alternatively make the buttons wrap around would be a simple change to this formula.) From the property sheet in the web interface builder, the user sets the "Source" property of the center image object to `=INDEX(B:B, A1)`. `INDEX` is a standard spreadsheet function that takes an array of cells (the first argument) and returns the cell at the given index (the second argument). The center image object now displays the photo at the index specified in A1, and will change when the user presses the previous or next buttons.

Lastly, the user has to create a list of thumbnails of all the photos. To do so, she drags a grid list to the bottom of the interface, drags an image object to the first item of the grid list, and set the "Source" property of that image object to be `=B1`. The system then populates the list to show photos from B2 to B5. The user wants the thumbnail displayed as the center image to be highlighted with a blue border. To do so, the user enters the following formula in the "Border" property of the thumbnail image:

```
=IF(THIS!Source=Image1!Source, "solid blue 1px", "none")
```

where `Image1` is the center image. The formula sets the border of a thumbnail to be blue when the source of the thumbnail is the source of the center image. The user now has finished creating a web application that shows photos in a slideshow interface, using only a few spreadsheet formulas.

4.4.2 ANIMATIONS

Continuing the photo slideshow example, suppose the user wants the width and height of the image to increase from 50 pixels to 100 pixels when the mouse enters it, and to go back to 50 pixels when the mouse leaves. She first sets spreadsheet cell C1 to be 50 and sets the "Width" property of the thumbnail image to be C1. She then types the following formula in C1:

```
=IF(Image2!State="hovered",
ANIMATE(C1, 100, 500), ANIMATE(C1, 50, 500))
```

where `Image2` is the thumbnail image. When the image is in the hovered state, this formula gradually increases C1 from its current value to 100 in 500 milliseconds. Otherwise, when not hovered, the formula gradually decreases C1 to 50 in 500 milliseconds. The user can then set cell D1 in the same way and make D1 be the height of the thumbnail to animate the height as well.

4.4.3 POSTING DATA

Gneiss can also create application that posts data back to a web data source using a POST web API. Suppose the user wants to create an application where she can type in a textbox and press a "Send" button to send the data to a web data source (instead of pressing the "enter" key as in the scenario in section 4.2). The user starts by creating a text box and a button in the output page. Then in spreadsheet cell A1, she types `=IF(Button1!State="clicked",TextBox1!Value, A1)` to fill A1 with the text box value when the button is clicked. She can then bind A1 to the value of the data parameter of the POST web API, which will send it to the data source.

4.5 LIMITATIONS AND DISCUSSION

This section discusses some limitations on how Gneiss' spreadsheet model supports programming interactive web applications. First, as mentioned previously in section 4.3.3, the current design of interactive properties does not allow the user to set the value of an input element programmatically, as all interactive properties are not editable and change only based on the user's interaction. This may be inconvenient

in several situations, such as when the user wants to initialize a GUI control to have a non-empty value, or to implement a "select all" checkbox that if checked will automatically check some other checkboxes. Gneiss could easily provide a new property that let user set the initial value of an input element. Another possible design to solve this problem is to make the input value and the user event separate properties, enabling the input value property to be calculated from event properties and other spreadsheet cells. For example, a checkbox could have a editable value property that by default is `=IF(THIS!State="Clicked", IF(THIS!Value="true", "false", "true"))` to have the default toggle behavior while allowing the user to customize the checkbox's behavior. But this also makes the system more complicated.

Also, Gneiss does not have a good way to let users combine sequences of events, such as detecting a drag-and-drop behavior. The user could do so by "streaming" events to spreadsheets (explained later in Chapter 6), but admittedly, it is quite awkward to program this in Gneiss. Prior spreadsheet tools that support programming graphical user interfaces have different ways to support this. For example, Forms/3 [11] supports a time model that can record an event and the time the event ended, enabling a programmer to reference past sequences of events, such as using a mouse down – mouse move – mouse up sequence to identify the behavior. InterState [71] uses a combination of states and constraints to specify transitions between different events to form a new behavior. So the user could let a GUI element have "drag" and "no drag" states such that in the "drag" state the element's position would be bound to the mouse position, and in the "no drag" state the element will stay at its current position. Transitions between "drag" and "no drag" are triggered by mouse down and mouse up events. Forms/3 and InterState were discussed in detail in section 2.3.1.

It is important to remember that Gneiss is designed as an end-user tool focusing on making web data more usable and useful. Therefore, while I have demonstrated in this chapter that with Gneiss the user can program many interactive behaviors, it is not the goal of this dissertation to support programming *all* kinds of custom graphics and interactive behaviors that one could think of with Gneiss. A programming system will inevitably get more and more complex when trying to support more things. As an end-user tool, I designed Gneiss to focus on supporting data-related interactive behaviors that I observed were essential in conventional data-driven applications, namely querying, sorting, filtering and visualizing data. In contrast, Forms/3 and InterState, while more powerful in creating custom graphics and interactive behaviors compared to Gneiss, introduce many new concepts with which spreadsheet users may not be familiar. I discuss more future work on extending Gneiss to better support programming interactive behaviors in section 8.3

The research in this chapter focuses on using spreadsheets to support programming data bindings and data-related interactive behaviors in a web application. Much future work can be done to enhance Gneiss to support styling a web application. For example, the web interface builder in Gneiss could have a "code" mode to let users directly edit the HTML and CSS code of the web application to give users more control (similar to what conventional web editors such as Adobe Dreamweaver do), or just let users import HTML and CSS files written elsewhere to use in Gneiss's interface builder. I discuss this further in section 8.7.

As described in this chapter's usage scenario in section 4.2, the user can choose to export a web application to use outside of Gneiss in any device that has a browser. Details on how Gneiss exports a web application are described in section 7.1.4. Future work includes increasing the usability of the exported web application on mobile devices, such as using responsive CSS frameworks to support different screen sizes, or to further support exporting mobile applications instead of web applications, which I discuss further in section 8.5.

4.6 CONCLUSIONS

In this chapter, I presented how Gneiss provides a novel way to create interactive, data-driven applications. Gneiss extends the spreadsheet model to unify the access of GUI elements, treating GUI element properties as spreadsheet cells that can use and be used in spreadsheet formulas. All GUI elements have interactive properties whose value changes based on how the user interacts with the element, enabling the user to program interactive behaviors using a "pull model" that is consistent with the current spreadsheet paradigm. Gneiss further supports once-around circular references and new functions such as `TIMER` to enable a variety of interactive behaviors to be programmed in spreadsheet languages. Combined with the spreadsheet model for using web services described in the previous chapter, Gneiss turns a spreadsheet into an intermediate platform that connects a web application and multiple web data services, and enables users to program two-way data flow between them all using the familiar spreadsheet mechanism. As shown by the various examples, Gneiss can be used to program a wide variety of web applications using a wide variety of web data sources.

CHAPTER 5 USING STRUCTURED HIERARCHICAL DATA[8]

Structured data formats such as JSON and XML are becoming more and more popular online due to the emergence of NoSQL databases provided by Web 2.0 companies such as Amazon, Facebook, Google and Yahoo [60]. JSON and XML are also the two dominant data formats for web services where the user can download various kinds of data such as music (e.g., Discogs), movies (e.g., RottenTomatoes), social networks (e.g., Facebook, Twitter), and finance (e.g., Yahoo, Bloomberg). In fact, programmableweb.com lists over 14,742 web services as of March, 2016, most of which return data in JSON or XML formats.

5.1 MOTIVATION, CHALLENGES AND CONTRIBUTIONS

Exploring and analyzing structured data has become a common task for many professional data analysts [4], and they often use programming languages such as R and Python[9]. Soon, many end-users may also have to work with JSON or XML data instead of comma-separated values (CSV) or Excel data as they do now. My dissertation extends spreadsheets, the most popular tool used by end-users for data analysis, to support using structured JSON data.

Often, structured data such as JSON and XML contain nested hierarchies. For example, Figure 5.2. at 1 shows a simple JSON object that describes an NBA team that has a list of players and for each player, a list of positions. This creates two levels of hierarchies, represented by the nested arrays. The team also has a list of coaches that is separated from the players' hierarchy. Current spreadsheet tools do not handle multiple nested hierarchical structures well, as a spreadsheet by nature is a two-dimensional table. Flattening a hierarchical document into a table will inevitably create many repeated values or empty cells, making further analysis of the data difficult. For example, Figure 5.2. at 2 shows the JSON object converted into a "long table" format where the hierarchies are expanded vertically. In this format, the player name "LeBron James" is repeated many times in order to correspond to each of the coaches' name, the player's position and NBA champion years. If there is a list of teams, it would be difficult to find out questions such as which player won the most NBA championships, as each player's name would be repeated a different number of times. The user would need to remove a lot of duplicated values before she could get the answer. Figure 5.2. at 3 shows the same data converted into a "wide table" format where the hierarchies are expanded horizontally. This format does not make things easier, as the players are now in different columns, and there

[8] The research in this chapter was also described in our publication at CHI'16 [22]

[9] See http://www.kdnuggets.com/2015/05/r-vs-python-data-science.html

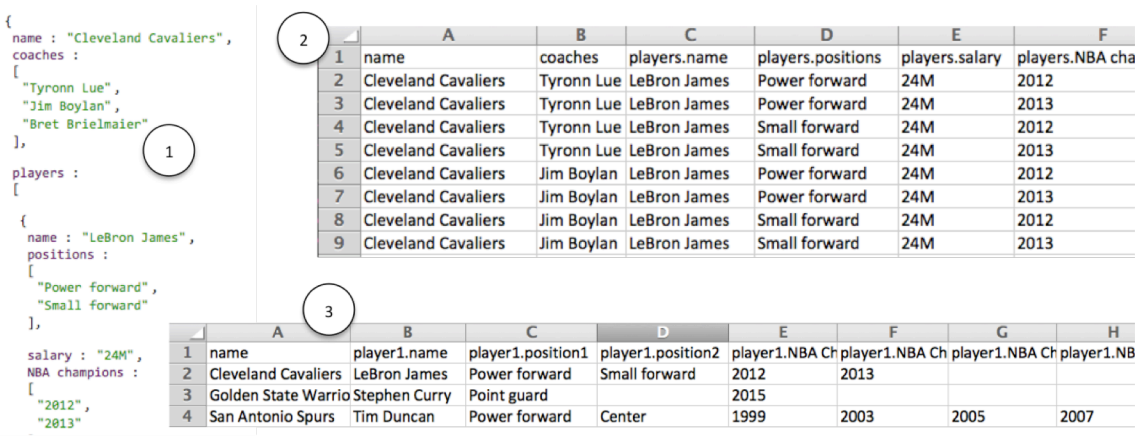
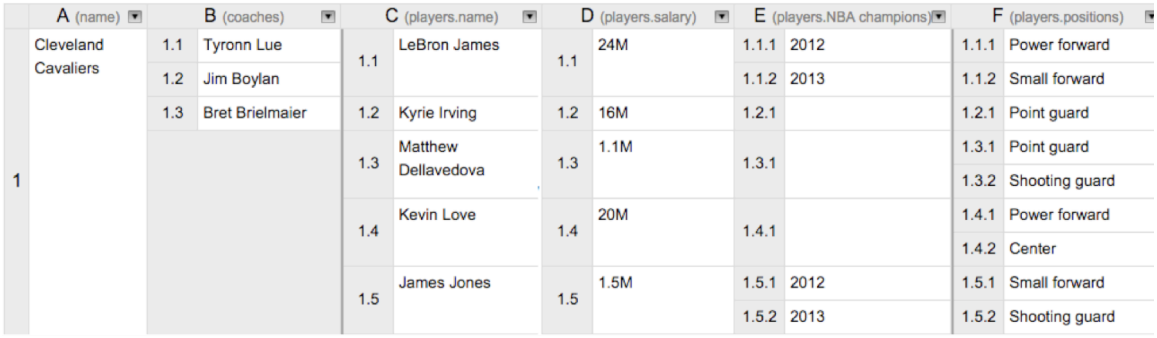


Figure 5.1. (1) A JSON object describing the Cleveland Cavaliers. The coaches and players are shown in two arrays. (2) Converting this JSON object into a "long table" spreadsheet format where the hierarchies are expended vertically creates a lot of repeated values, whereas (3) converting it into a "wide table" spreadsheet format where the hierarchies are expended horizontally creates a lot of columns with empty cells.

are many empty cells in the columns. The format is also very difficult to read, as the table has a lot of columns and requires the user to scroll quite far horizontally to see everything. This JSON document includes each player's salary data. However, there is no easy way for the user to get the highest paid player in each team in either table format using spreadsheet mechanisms. This is because the spreadsheet does not support selecting and manipulating data (such as sorting and filtering) using the hierarchical structure.

To address these problems, my dissertation introduces a spreadsheet model for using structured hierarchical data. This model makes three main contributions. First, it contributes a new method to visualize hierarchical data in a spreadsheet that lets users reshape and regroup data easily through interaction techniques. Data can be dynamically visualized into nested cells (Figure 5.2.) or reshaped into a flat table based on the relative hierarchical relationship among spreadsheet columns. This feature enables the user to easily explore different groupings of data using any JSON fields. It also allows the user to calculate various kinds of summaries of data using the familiar spreadsheet functions without having to learn new concepts such as pivot tables or SQL queries.

This model makes a second contribution by extending the familiar spreadsheet mechanisms for manipulating table data, namely spreadsheet languages, sorting, filtering, and autofilling (select-and-drag), to support hierarchical data as well. Gneiss generates nested row labels for hierarchical data to allow users to select data using its structure using the familiar spreadsheet language syntax. The nested tables also enable sorting and filtering of the data using the hierarchy. For example, in Figure 5.2. , each team is a row, and each team's players form a nested table within a row. To view the highest paid player in each team, the user can sort by the salary



	A (name)	B (coaches)		C (players.name)		D (players.salary)		E (players.NBA champions)		F (players.positions)	
1	Cleveland Cavaliers	1.1	Tyronn Lue	1.1	LeBron James	1.1	24M	1.1.1	2012	1.1.1	Power forward
		1.2	Jim Boylan					1.1.2	2013	1.1.2	Small forward
		1.3	Bret Brielmaier	1.2	Kyrie Irving	1.2	16M	1.2.1		1.2.1	Point guard
				1.3	Matthew Dellavedova	1.3	1.1M	1.3.1		1.3.1	Point guard
										1.3.2	Shooting guard
				1.4	Kevin Love	1.4	20M	1.4.1		1.4.1	Power forward
										1.4.2	Center
				1.5	James Jones	1.5	1.5M	1.5.1	2012	1.5.1	Small forward
								1.5.2	2013	1.5.2	Shooting guard

Figure 5.2. The JSON object in Figure 5.1 at 1 shown in Gneiss. Gneiss visualizes the hierarchies of data using nested cells with nested row labels to allow users to select data by its structure. In this JSON object, the team coaches and players are in two different arrays. Therefore, Gneiss uses a thin grey line between them (column B and C) to show that they are not hierarchically connected (same as the player's positions and NBA champion years in column E and F).

column to bring each team's highest paid player to the top of its nested table, and filter to see the first item only in each nested table. The user can also insert a new column that has the same structure as any column in the spreadsheet, and create new hierarchical data using spreadsheet languages and autofilling.

Finally, our tool contributes a new method to join hierarchical data from multiple sources based on common fields. While hierarchical documents could be turned into flat tables and joined using conventional methods (such as in [25]), our method operates directly on hierarchical objects, connecting two trees without flattening them. This creates in a new combined hierarchical object that can be reshaped, regrouped, sorted and filtered using its structure as regular hierarchical objects in Gneiss.

The user further can use the right pane in Gneiss to make visualizations that take hierarchical data such as a treemap [45] (see section 5.4.1), or show the data in a web application using nested lists (see section 4.3.1.2).

In the rest of the chapter, I will explain those features in detail, and present examples to show the model's ability to let users explore and analyze hierarchical JSON data. I will also describe a lab study where I recruited intermediate spreadsheet users to use Gneiss or Microsoft Excel to complete five data exploration tasks using two hierarchical JSON documents. I also recruited a separate set of people who were professional programmers familiar with JSON data to use JavaScript or Python to complete the same tasks. In summary, Gneiss helped spreadsheet users complete the tasks nearly two times faster than using Excel. Gneiss even helped spreadsheet users complete four of the five tasks faster than professional programmers. Participants rated our study tasks highly realistic and similar to what they do in their own work in real life. Based on participants' feedback and our observation on how they completed the tasks, I discuss the

Figure 5.3. Two example JSON objects used in the usage scenario. The left JSON object describes a conference paper. The right JSON object describes a paper session in a conference, with each paper listed in order, using its ID.

strengths and limitations of Gneiss in helping spreadsheet users use hierarchical data, and some possible directions for future work.

5.2 USAGE SCENARIO

Here I present another usage scenario to give an overview of the novel features for using and exploring hierarchical data. In this scenario, Ally, a graduate student studying HCI, gets two JSON documents about the CHI'15 conference. Ally is a fluent spreadsheet user familiar with spreadsheet functions. Figure 5.3 shows an example object from each file. The first file, papers.json, stores all accepted papers. Each paper is an object and has 7 fields (Figure 5.3 at the left): ID, paper_title, abstract, keywords (an array), type ("long" or "short" meaning a long or short paper), award ("none", "bp" for a best paper, "hm" for a honorable mention paper) and authors (an array with each author being an object and having a name, institution, city and country field). The second file, sessions.json, stores all the paper sessions in the conference. Each session is an object and has 6 fields (Figure 5.3 at the right): session_title, room, day, time, chair and submissions. The submissions field is an array that contains an ordered list of paper IDs that are presented in the session. Ally is interested to know which institution has the most accepted papers and also wants to determine which paper sessions she should attend during the conference. She uses Gneiss to explore the data.

Ally loads the two files into two tabs of the left pane using the "Choose file" button. She first tries to answer the question of which institution has the most papers. In

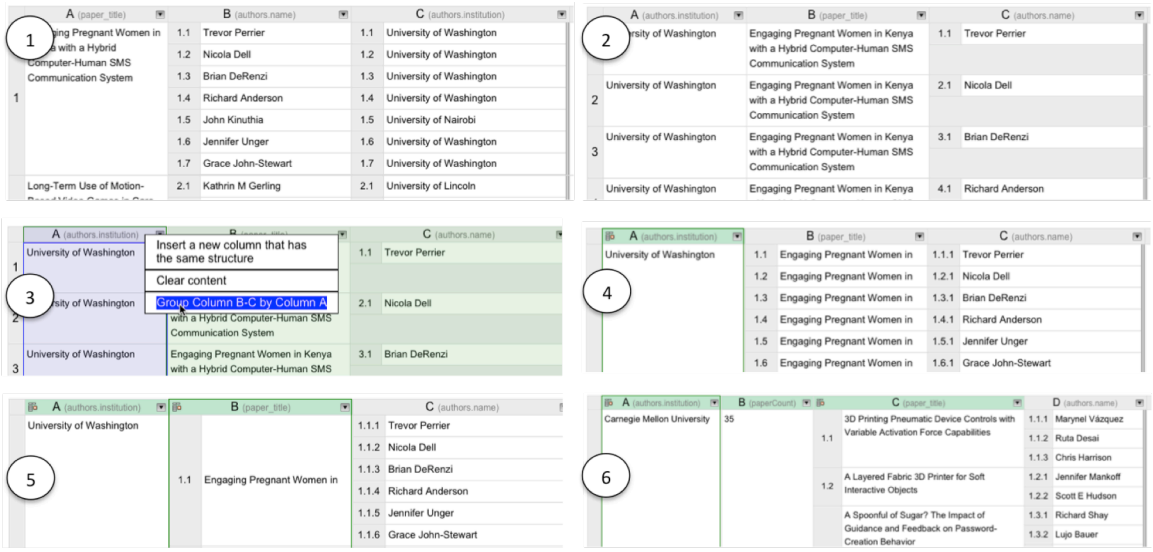


Figure 5.4. Six tables showing how the user can compute summaries of a JSON file of conference papers. (1) The data initially is indexed by paper titles. Each paper has multiple authors. Each author has an institution. The user regroups the data by institutions by dragging the institutions to the beginning of the table (2), right-clicks column A and chooses "Group Column B-C by Column A" (3) to merge rows that have the same value in column A (4). Column B now has many repeated values because a paper can have many authors from the same institution. (5) The user groups the data again by column B to merge repetitive paper titles within an institution. Lastly, (6) the user inserts a new column at column B, enters =COUNT(C1) in B1 to get the paper count for the first institution, and autofills the value for the rest of the institutions. She sorts the data by column B to bring the institution that has the most papers to the top.

Gneiss, the user starts by extracting fields that she thinks are relevant to her task. Ally extracts the paper titles, author names and institutions to spreadsheet columns A – C respectively (Figure 5.4 at 1) using drag-and-drop. The system recognizes that column B (author names) came from a child field of column A (paper titles). Therefore, it puts author names in nested tables in column B so they are in the same row as their corresponding paper in column A. As for column C (author institutions), the system recognizes that the data are in the same hierarchy level in the document as column B, so it creates the same nested tables as column B and puts an author's name and institution in the same row.

This view lets Ally easily see how many authors and institutions each paper has, but Ally wants to know which institution has the most papers. She needs to regroup the data using the institution field. To do so, she drags the institution data from column C to A. In our tool, data that are used for grouping must be at the beginning of the table. The system reorganizes the data accordingly (Figure 5.4 at 2): column A now shows a flattened list of all of the institutions fields. Data in column B (paper titles) now come from a parent field of the data in column A (author institutions) and thus the system repeats the parent values to let the child and parent again be in the same row. Finally, the system recognizes that data in column C (author names) come from a child field of column B so it shows the data in nested cells. Column C is also

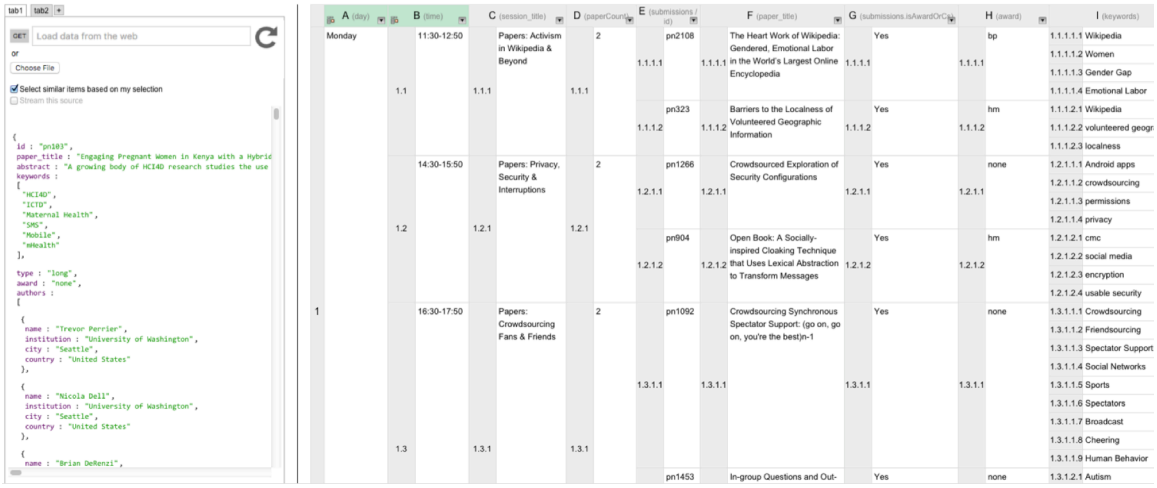


Figure 5.5. A screenshot showing the result spreadsheet of the second task in the usage scenario. The left pane is where the user loads and views a JSON file (through a URL bar for online data or a "Choose file" button for local data). The spreadsheet is a custom conference schedule created by the user. The spreadsheet shows for each time slot, the session that has the most number of award papers plus papers that have "crowdsourcing" in its keywords. The data is grouped by day (column A) and time (column B). Column C shows the title of the session that the user should go to. The number of award plus crowdsourcing papers of that session is in column D, with the details of the papers in the rest of the columns.

constrained by column A: only the author name in the same object with the author institution in the same row is shown.

Ally then selects column A, right clicks and chooses "Group Column B-C by Column A" (Figure 5.4 at 3). The system merges rows that have the same values in column A, and column B becomes nested (Figure 5.4 at 4). She can see that many papers are repeated multiple times. As column C shows, this is clearly because a paper could have multiple authors from the same institution. To get rid of duplicate values in column B, Ally groups the data by column B using similar methods. Now each spreadsheet row shows an institution, all papers of the institution, and all authors who wrote the paper and are from the institution (Figure 5.4 at 5).

With this table, Ally can easily see the number of papers of each institution. She inserts a new column next to column A and names it "paperCount". Column B is now a blank column and the paper titles and author names are pushed to columns C and D. Ally enters `=COUNT(C1)` in B1 to count the number of papers of the first institution, and uses drag-down autofill to fill in the paper count for the rest of the institutions. She sorts all the data by column B to bring the institution that has the most papers to the top (Figure 5.4 at 6). With a few mouse clicks and a spreadsheet function, Ally gets the answer she wants from a messy JSON file.

Ally's next task is to come up with a schedule that tells her what sessions to attend at the conference (Figure 5.5). She is interested in papers about crowdsourcing and also all the award papers. She decides that for each time slot, she wants to go to the

session that has the highest number of award papers plus papers that have “crowdsourcing” as one of the keywords. Since session and paper information are in two different files, she needs to *join* the two documents together. Ally opens a blank spreadsheet and first extracts the fields she needs from the session file, which are the day, time, title, and paper submissions (listing their IDs) for each session, from the left pane into spreadsheet columns A-D. She then switches to the paper file and extracts the paper ID, title, award and keywords fields into columns E-H. She then selects column D and E that are the paper IDs from the session and paper files, right-clicks and chooses “Join The Two Tables on Column D=E” (Figure 5.6 at the top).

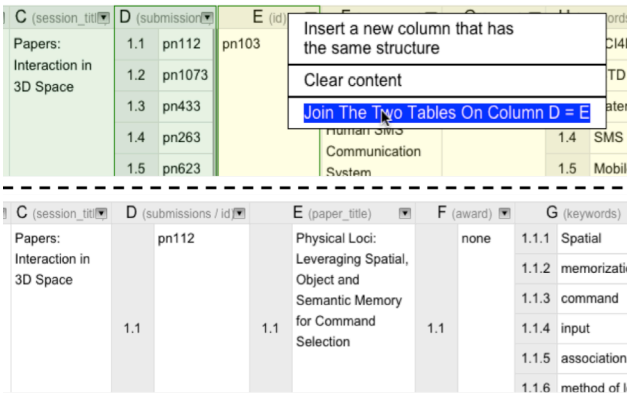


Figure 5.6. The user joins two tables by selecting a common column in each table and clicking the “Join” option from the right-click menu. The common columns (here, columns D and E) are combined into one column after the tables are joined.

The system connects the two tables by joining the two ID columns. It creates a combined hierarchical table containing three levels of hierarchies (Figure 5.6 at the bottom). The user can manipulate this table such as reshaping it by moving columns or sorting and filtering values the same way as if the data had come from a single source. Here, Ally wants to know the total number of award and crowdsourcing papers in a session. She inserts a new column next to the paper title in column F (Figure 5.7), naming it “isAwardOrCs”. The award and keyword column are pushed to column G and H. Ally then enters this formula in cell F1.1:

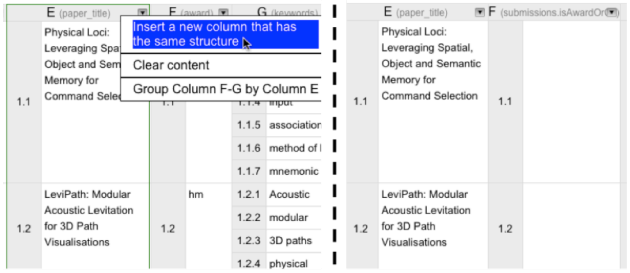


Figure 5.7. The user can insert a new blank column that has the same structure as a selected column.

```
=IF(OR(OR(G1.1=”bp”,G1.1=”hm”), COUNTIF(H1.1,“crowdsourcing”)>0),
                                “Yes”,“No”)
```

The formula returns “Yes” if a paper has an award (best paper (bp) or honorable mention (hm)) or has at least one keyword being “crowdsourcing”. Ally then computes this value for all papers using autofill, and filters the data to show only the papers that have “Yes” in this column.

Ally inserts a new column next to the session title column (Figure 5.5 at column D), uses the `COUNT` function to compute the number of “Yes” papers for the first session,

and autofills the rest of the sessions. Finally, Ally groups the data by the day and time to make it more readable. Within each time slot, Ally uses sorting and filtering to show the session that has the highest count number. Now she gets her custom schedule! The final spreadsheet is in Figure 5.5.

5.3 KEY FEATURES FOR USING HIERARCHICAL DATA

Here I describe the key features for using hierarchical data in Gneiss.

5.3.1 DISPLAYING HIERARCHICAL DATA IN SPREADSHEETS

The design of Gneiss's hierarchical data display has gone through several iterations. In an early version [20], the user could extract an entire JSON object into a column, and the object was visualized as nested tables with nested column headers. **Error! Reference source ot found.** is an example of the old design showing movie data. The user extracted the entire abridged_cast object for each movie to column B. Abridged_cast is an array, each item in the array is a JSON object that has three fields: name, id, and characters. The characters field is another array where each item in the array is an object that has a field called character. To refer to the cell "Ewan McGregor" in the nested table, the user could enter `B1.A2.A1`, which is the cell name sequence starting from the root cell to the target cell. As in a conventional spreadsheet, when typing a spreadsheet formula the user could also click on a cell to insert its name.

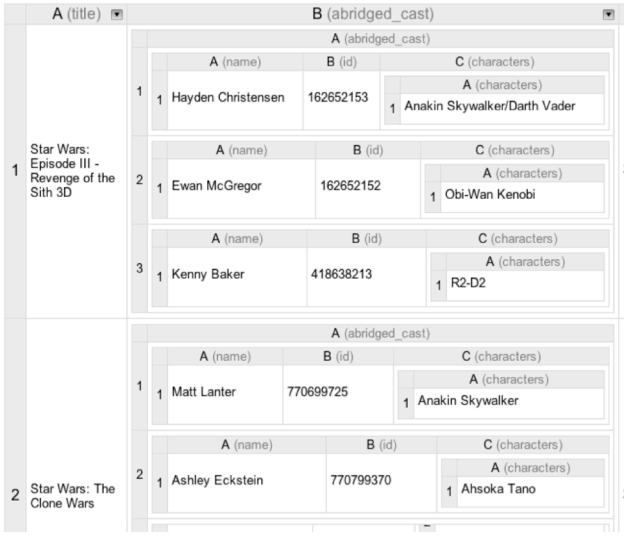


Figure 5.8. An old design of Gneiss's nested table.

This design, while able to show the structure of the data, has several disadvantages. First, the nested column headings would take a lot of screen space if the object has complex structure (even in the example in **Error! Reference source not found.** here the data only have two levels of structure, the nested headings already take a lot of space). Second, counting the nested cell names also becomes difficult when the nested structure gets deeper. For example, in **Error! Reference source not found.**, he name for the cell that has the value "R2-D2" is `B1.A3.C1.A1`. But after running a few informal pilot tests, I found that with this spreadsheet design, users tended to think that the cell name would start with A1, which is the leaf cell that directly wraps the value "R2-D2", as that was the cell that they first saw. Instead, in this design, the user would have to trace back to the cell's ancestors to the root and go

down again to compose a cell name, which is not intuitive and it is easy to get lost. Finally, an observation I had was that often when doing data analysis, the user would only need to use a few fields from a JSON document at a time. Since visualizing the entire JSON document object would often generate large and complex nested tables that are difficult to use, a better strategy might be to focus on visualizing only the fields that the user wants and to support new ways to use and manipulate those data. My observation was that when a JSON object is properly formatted, it is quite understandable by end-users, especially to users with the relevant domain knowledge. For example, in my user study where I showed JSON data about conference papers (Figure 5.3) to graduate student participants, everybody could understand the meaning of the properties and the data structures even when they were not familiar with JSON syntax.

So I redesigned how users could use hierarchical data in Gneiss. In the new design, Gneiss's interface lets the user start using a document by extracting to the spreadsheet just the value fields (fields that are strings, numbers or Boolean values) that she thinks are relevant to her task. This allows the visualization method to run on a smaller set of data and thus generate cleaner tables. The new nested table design replaces nested headings with nested row numbers that are placed right next to each cell (see Figure 5.9 for a comparison), so the user does not need to look for the parent structure to get the cell name. The new design also wastes much less space for cell address letters.

	A (movies.titl	B (movies.abridged_cast.nam		C (movies.abridged_cast		D (movies.abridged_cast.charact	
1	Star Wars: Episode II - Attack of the Clones 3D	1.1	Hayden Christensen	1.1	162652153	1.1.1	Anakin Skywalker
						1.1.2	Darth Vader
		1.2	Natalie Portman	1.2	162652154	1.2.1	Padme Amidala
		1.3	Ewan McGregor	1.3	162652152	1.3.1	Obi-Wan Kenobi
		1.4	Samuel L. Jackson	1.4	162652156	1.4.1	Mace Windu
		1.5	Christopher Lee	1.5	162652385	1.5.1	Count Dooku
						1.5.2	Darth Tyranus
2	Star Wars: The Clone Wars	2.1	Matt Lanter	2.1	770699725	2.1.1	Anakin Skywalker
		2.2	Ashley Drane	2.2	770724065	2.2.1	Ahsoka Tano
		2.3	James Arnold Taylor	2.3	178810504	2.3.1	Obi-Wan Kenobi
		2.4	Catherine Taber	2.4	771037991	2.4.1	Padme Amidala
		2.5	Samuel L. Jackson	2.5	162652156	2.5.1	Mace Windu

Figure 5.9. The new design of Gneiss's nested table, showing similar data as in Figure 8.

Having the user extract the fields instead of visualizing the entire JSON file also ensures that the extracted fields are somewhat related, as judged by the user. Gneiss's visualization method then focuses on using the hierarchical relationships among data in adjacent spreadsheet columns to support reshaping and regrouping a hierarchical object by any field through interaction techniques.

5.3.1.1 Definition of hierarchy

Gneiss targets JSON data, which is organized into objects that contain named fields, each of which can be an object, an array, or a value that is a string, a number, true/false or null. In my current implementation, I treat each JSON array as a hierarchical tree. Each item in an array creates a branch and a node in the tree.

Fields in the same array item are considered being in the same level of the tree and thus are siblings. For example, in the usage scenario, *papers.json* is a big array where each item in the array describes a paper (Figure 5.3 at left). It can be seen as a tree where each paper forms a branch. Fields in the same paper item are considered as siblings, such as the title field and type field. If an array item has a field that is also an array, that field creates a subtree in a branch, thus the tree grows a new level. For example, the authors field in a paper item is an array, so Gneiss creates a subtree under a paper item where each author in the array becomes a new branch, and the root tree grows a new level (so it now has three levels). The keywords field in a paper item is also an array, so it forms another subtree under a paper item where each keyword creates a new branch. The root tree, however, still is three levels deep since the keywords tree and authors tree start at the same level and have the same depth. If a field A can reach another field B by only going up towards the root through branches, field B is then an ancestor of field A, and field A is then a descendent of field B. For example, the paper title field can be reached from the author name field by traveling up through a branch. Therefore paper title is considered an ancestor of author name, and author name is a descendant of paper title. A keyword item, however, cannot be reached from the author name field by going only up or down branches. Therefore, there is no specific hierarchical relationship between the two fields.

My current implementation expects each array item to have a similar structure. This hypothesis is true for most of the database and web service data I have seen because the data usually follows a predefined schema. If there is a missing field in an array item, that field is shown as a blank cell in the spreadsheet. While different types of data could specify hierarchies differently, I believe that our method could be extended to support those data as well. I will discuss this further in the limitations section below.

5.3.1.2 Terminology

In the rest of this chapter, I use the term *hierarchical table* to refer to a set of adjacent columns that have fields extracted from the same tree in a source document. There can be multiple hierarchical tables in a spreadsheet. I use the term *nested table* to refer to a set of inner cells that have the same direct parent cell. Each inner cell creates a *nested row* in the columns that are in the same hierarchical table to its right. A nested row can contain a value or another nested table.

5.3.1.3 Visualization algorithm

Gneiss introduces a new visualization method that uses the relative hierarchical relationships between data in adjacent spreadsheet columns to visualize JSON fields extracted by the user in a spreadsheet. The method allows the user to create

different views of the same data, flattening it or regrouping it, simply by changing the order of the data in spreadsheet columns through drag-and-drop. Here I describe this visualization method.

For each column in the spreadsheet, the system records if its data were extracted from a hierarchical document and if so, the name of the document and the path to the data. The system then scans through the columns to determine which columns form a hierarchical table. In a hierarchical table, the leftmost column is always shown as a flat (regular) column. For example, in Figure 5.4 at 1, column A is a flat column where each cell in the column stores a paper title string. For each of the rest of the columns in the hierarchical table, the system starts by examining its immediate left column.

1) If the column immediately to the left comes from **an ancestor field**, the system puts data in this column in inner cells in the same row with the ancestor value. Those inner cells form a nested table. Each cell in a nested table creates a nested row that has a row label `parentLabel.thisIndex`. For example, in Figure 5.4 at 1, a paper title field is an ancestor field of the author name field. Thus when displaying cell B1, the system puts the author names that belong to the paper title in A1 into inner cells in row 1. The author names thus form a nested table inside cell B1. Each cell becomes a nested row and has a row label 1.1 – 1.7 as there are seven items in the first paper's authors array. The order of the values is the same as their order in the array in the source document unless sorted by the user.

A nested row can further contain other nested tables. For example, in Figure 5.5, due to the user's joining and grouping operations, each paper is already in a nested row. Thus each paper's keywords (column I) are put in a nested table inside each nested row with the paper, creating additional levels of structure.

To decide what values to put in a nested table, the system will keep checking until it reaches the leftmost column to see if there is a column whose data is from a descendent field of the immediate left column and is also in the same branch with the current column. If the system finds one, it stops looking and puts only the values that are in the same hierarchical branch with that descendant field into the nested table. If the system does not find a column that fits the criteria, all the descendant values of the immediate left column are put into the nested table. Using Figure 5.4 at 2 as an example, when visualizing cell C1 (author name), the system first looks at B1's value, which is the title of the first paper. The system knows that paper title is an ancestor of author names, thus it generates a nested table in C1. To decide which author names from the first paper should be put in the nested table in C1, the system moves to the next left column and checks A1's value. A1 is the institution of the first author of the first paper. Therefore, C1's value is constrained by A1: the system puts only the name of the first author of the first paper in C1. If A1 were an

empty column instead, the system would put all authors' names of the first paper into the nested table in C1.

2) If the immediate left column comes from **a sibling field**, the system copies the structure of the immediate left column and puts the values in the same row with its siblings. For example, in Figure 5.4 at 1, column B (author name) and column C (author institution) are siblings. Therefore the system copies column B's structure to C, and puts the institution that is in the same object in the source document with the author name in cell B1.1 into cell C1.1, and so on.

3) If the immediate left column comes from **a descendant field**, the system copies the structure of the immediate left column and for each row puts the ancestor value that is in the same branch in the hierarchical tree with the descendant. For example, in Figure 5.4 at 2, when visualizing column B (paper title), the system checks column A (author name) and finds that it comes from a descendant field. Therefore it copies column A's structure (which is just a flat column) to column B and fills in B1 with the ancestor paper title of the author name in A1, and so on. As Figure 5.4 at 2 shows, an ancestor value could be repeated multiple times, as multiple descendants could have the same ancestor.

4) If the immediate left column does not have any hierarchical relationship with the current column, the system checks the next leftmost column for the same relationships as described above, and so on. If a relationship is found in a column that is not the immediate neighbor, the system displays a thin gray line between this column and its immediate left column to show that the two are not connected hierarchically (such as column B and C in Figure 5.2.). If the system could not find any columns in the hierarchical table that relate to this column, it treats this column as the start of a new independent hierarchical table which is shown by separating the column with a thick gray line (such as column D and E in Figure 5.6 at the top).

Using this method, the nesting level of a hierarchical table always increases from left to right. Every time when the user drags a column to a different location, the system recomputes the hierarchical tables that are affected and updates the interface. Internally, every hierarchical table is stored as a JSON object. The system dynamically changes the structure of the JSON object using the column locations and the grouping, sorting and filtering rules. The implementation is discussed further in Chapter 7.

5.3.2 MANIPULATING HIERARCHICAL DATA

An important goal of Gneiss's spreadsheet model for using hierarchical data is not just to provide a way to *display* hierarchical data in a spreadsheet, but also allow users to *manipulate* hierarchical data using the familiar spreadsheet mechanisms.

This makes Gneiss different from systems that only visualize hierarchical data (e.g., [8]) and systems that use other languages such as SQL to support manipulating hierarchical data (e.g., [25]). Gneiss's spreadsheet model supports very natural and easy-to-use interactions for regrouping hierarchical objects by arbitrary fields, using hierarchical data in formulas, creating new fields in a hierarchical object, sorting and filtering data by its hierarchical structure, and joining multiple hierarchical objects. Behind the scenes, those manipulations operate directly on JSON objects, changing their structure, adding new fields and sometimes even creating new objects, as described in detail later in section 7.2. My user study (section 5.5) showed that these interactions are usable. The next sections explain them in detail.

5.3.2.1 Extracting hierarchical data and reshaping them

As described earlier, I extended the familiar drag-and-drop gesture to support two features that help people manipulate hierarchical data in Gneiss. The first is to extract desired fields from a JSON document to a spreadsheet. As mentioned earlier in section 5.3.1, the decision to let users select the desired fields to be shown in a spreadsheet is based on the observation that often not all the information in a JSON document returned from a web service or database is relevant to the user's task. Also, in my user study, participants seemed to have no trouble selecting fields that they wanted from a JSON document even if they did not have prior experience using JSON. The drag-and-drop gesture for extracting desired fields from a document replaces the need to write textual queries such as `SELECT` statements in SQL or other document query languages such as XPath.

Second, the user can also use drag-and-drop to change the order of the data in spreadsheet columns to generate different views of the data. As described in the previous section, putting an ancestor field before (to the left) of a descendent field will put the descendants in nested tables to create a structured view (such as column A and B in Figure 5.4 at 1), and putting a descendant field before the ancestor field will cause the ancestor to be repeated, creating a flat view (such as column A and B in Figure 5.4 at 2). This by itself is useful as it allows the user to structure or flatten data based on her needs. For example, in Figure 5.2. , the user puts the team's names before the coach's names and creates a hierarchical view to see the coaches for each team. The user can also switch the order to drag the coach's names in front of the team names. In that case it creates a flat table with each coach's name and his/her team in a row. The user can then sort the coaches to view them alphabetically. Currently, there are no query languages that allow users to restructure hierarchical data like this. The user would have to write their own code to get the appropriate fields and reorganize them.

5.3.2.2 Regrouping the data by arbitrary fields

The user can further regroup a hierarchical table in the spreadsheet using any fields. As described in the scenario in section 5.2, the user would first move the columns that hold the fields by which she wants to group the data to the beginning (left) of the table. This causes the table to be first flattened by the selected fields, as the first column of a table is always flat and then the hierarchy builds up to the right. The user can then select the grouping columns, right-clicking and choosing the "group by" option from the menu. The text for the "group by" menu item is generated dynamically based on what the user selects (see Figure 5.4 at 3 for an example). When the user hovers the mouse over the "group by" menu item, the system highlights the columns used for grouping in purple and the columns being grouped by in green to help the user identify the range (Figure 5.4 at 3). The "group by" operation combines rows that have the same values in the grouping columns (note that it does not sort the rows first – rows that have the same values are merged based on the topmost row in which the value appears). Columns that are to the right of the grouping columns are merged into nested tables. Internally, the grouping operation merges nodes in a JSON object tree through selected fields (the selected grouping columns), and puts the rest of the fields in the nodes in the next level of structure.

A column used for grouping will have a green column label. The user can cancel a grouping anytime by clicking on an icon at the top left of the grouping columns (Figure 4 at 4). As shown in the scenario, the visualization algorithm and the grouping feature enable users to easily experiment with different ways to view and group data through simple interaction techniques. In conventional spreadsheets, to group data, the user would need to use pivot tables which use a different interface and takes the user away from the data. In other cases, the user has to write complex SQL queries using the `GROUP BY` keyword to specify grouping criteria, and the `GROUP BY` keyword in SQL must be used together with some aggregating functions since SQL can only return flat tables. Instead, Gneiss supports hierarchical tables and allows users to group the data without using aggregated functions to flatten them first. Also, as I discussed in the beginning of this chapter in section 4.1, converting a hierarchical document into a flat table will often create many empty cells or repetitive data that require the user to do lots of data cleaning and filtering before she can use pivot tables or SQL to process the data. Gneiss does not have this problem.

5.3.2.3 Using nested data in spreadsheet formulas

As introduced in section 3.4.5, Gneiss introduces an extended spreadsheet language for selecting values in the nested cells. A nested cell can be referenced as a regular cell using its column and row label. For example, in Figure 10 at 1, `B1.1` selects the value "Primary education" which is the first major sector of the first country. Our

language also supports selecting multiple values. Conventional spreadsheet's ":" operator for specifying the start and end cell of a range selection still works in our tool. For example, `C1.2:C2.1` in Figure 10 at 1 selects 4 values (26, 16, 12, 70). The user can also use the parent row label to select all cells in a nested table. For example, `B1` selects all sectors of the first country in Figure 10 at 1. Finally, our language also includes a wildcard character (`*`) that can be used in any nested row index to further assist hierarchical selection. For example, `B*.1` in Figure 10 at 1 selects the first primary sector of all the countries. Selections that return multiple values are put in a flat array and thus can be used by many conventional spreadsheet functions that take a list of values as inputs, such as the familiar `COUNT` and `SUM` (if not used with functions that accept multiple values, the system will raise an error as in conventional spreadsheets). This language enables Gneiss to go beyond a read-only tool and allows the user to compute new values using the hierarchical data, as shown in the scenarios.

5.3.2.4 Inserting new data into a hierarchical table

There are many situations where the user may want to add new data to a hierarchical table. For example, in the usage scenario in section 5.2, when finding out which institution has the most papers, Ally adds a flat column to compute the number of papers for each institution using a spreadsheet formula. She may also want to add a nested column next to the author names and manually enter an author's email address. Our tool lets users insert different structures of columns by first selecting a column in the table that has the desired structure, and then right-clicking and choosing "Insert a new column that has the same structure" (Figure 5.7 at the left). A new column will be inserted to the right of the selected column. If the selected column is a regular flat column, the inserted column is also flat; if the selected column is a column containing nested tables, the inserted column will have the same tables but with empty cells (Figure 5.7 at the right).

The newly created column can be moved and used for grouping, sorting, filtering and joining just like any other column. To enable this, every time the user selects an existing column to insert a new column into the spreadsheet, the system creates new fields in the same hierarchical level with the data in the existing column in an internal copy of the source JSON document. For example, in the first example scenario, the user selects the author institution column and inserts a new column called "paperCount" for each author institution (Figure 5.4 at 6 at column B). The system then creates a new field named "paperCount" in each author object (so it is in the same hierarchical level with the author institution), inserts a new column at B, and uses both column A and B for grouping. So when the user edits cell B1, internally she changes the value of the paperCount fields in all author objects where the institution field is "Carnegie Mellon University". Later if the user decides to

change the order of the columns, the system can reference this copy of the source JSON document to decide how the new data should be visualized.

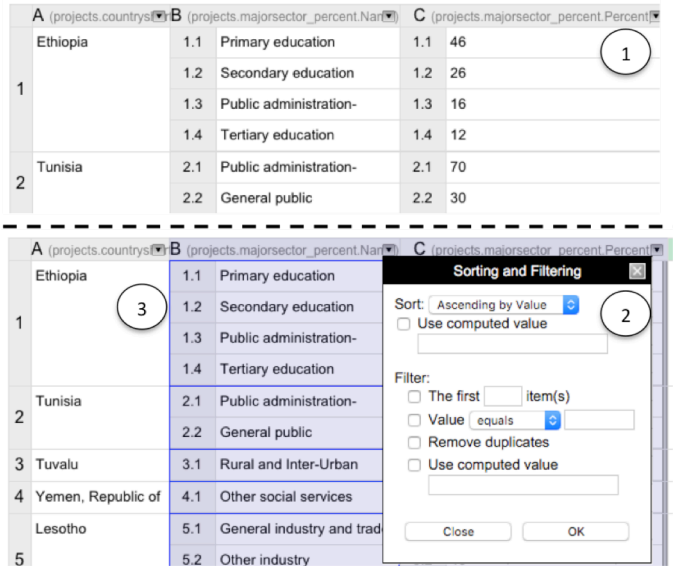


Figure 5.10. (1) shows countries funded by the World Bank. Each country has a list of major sectors and the percentage of funding each sector received. The user can sort and filter the data by any column using a dialog box (2). When sorting and filtering on a nested column, the operation is executed within each nested table, marked in the spreadsheet with blue borders (3). The system highlights the affected columns using a purple background when the dialog box is open.

5.3.2.5 Sorting and filtering data using its hierarchical structure

The user can choose to sort and filter a hierarchical table using any column by clicking on the arrow icon at the top of the column to open a dialog box (see Figure 5.10). Internally, sorting and filtering operations are essentially reordering and cutting branches of a JSON object tree. Transforming to the user interface, sorting and filtering by a nested column are executed within each nested table in that column. The operation will affect columns that have the same or deeper nested level, but will not affect columns that are in the upper levels of the hierarchy. To help the user identify the nested tables and the affected columns, when the sorting and filtering dialog box is open, the system highlights the affected columns in purple and highlights each nested table in the column using a blue border. For example, in Figure 5.10, sorting on column B will affect columns B and C but will not change the data in column A, as column A is an ancestor of column B. If columns A and B had the same nesting level, then they would both be sorted. Each nested table in column B, outlined using a blue rectangle, is sorted internally. The same rule applies for filtering. Filtering to see the first *n* items will show the first *n* items in each nested tables. Duplicated values are also calculated within each nested table.

Hierarchical sorting and filtering can be useful in many situations. For example, in the usage scenario in section 5.2, it was used to create a schedule that for each time slot shows the session that has the most award and crowdsourcing papers, Ally uses hierarchical filtering to get rid of papers within a session that she is not interested in, and uses both hierarchical sorting and filtering so that each time slot shows only the session that has the most award and crowdsourcing papers. In conventional spreadsheets, since the spreadsheet only accepts flat tables, it is often tricky and sometimes tedious to achieve the same results. For example, to get the session that has the highest number of desired papers in each time slot, in conventional

spreadsheets the user can first sort the table by the number of desired papers of a session. This brings the sessions that have the most desired papers *across all time slots* to the top. The user can then sort again by time slots. That will reorganize the data to let the sessions that are in the same time slot be put together, and within each time slot the session that has the most desired papers will be at the top (because of the first sort). But then the user cannot use filtering to keep only the top session in each time slot. She has to manually delete the unwanted sessions. Or, to use methods other than sorting and filtering, the user could put the data into a pivot table, or create additional data (such as a clean list of unique time slots) and use `LOOKUP` or similar functions to calculate the results. Again, my design in Gneiss removes the need to use pivot tables and advanced spreadsheet functions such as `LOOKUP` for cross-referencing. My user study (section 5.5) showed that the design of hierarchical sorting and filtering was understandable by the participants and was critical in helping them complete the tasks.

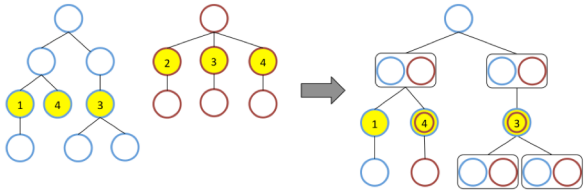


Figure 5.11. An example of joining two hierarchical objects. The two trees are joined by the yellow nodes. (Left) The blue tree has the yellow nodes in a deeper hierarchy level than the red tree. (Right) The join operation preserves the blue tree's structure and connects the red tree to the blue tree through the matched yellow nodes (shown as the yellow nodes that have two rings).

5.3.2.6 Joining hierarchical data

Gneiss supports joining hierarchical tables based on the columns with common values selected by the user in each table. Like the operations described above, the joining operation also works directly on hierarchical objects, connecting two trees through common nodes. The system first creates a duplicate JSON object for each table that has the same hierarchical structure as its source JSON document but contains only the fields that are visible in the spreadsheet. The system then checks the hierarchical level of the selected joining fields in the two duplicate objects. Gneiss preserves the object that has the joining fields in a deeper level of the tree, and connects the other object that has the joining fields closer to the root to the preserved object (see Figure 5. for an example). Based on this rule, the preserved object is not necessarily the left object in the spreadsheet, since whether an object is preserved is decided by the hierarchical level of the join fields in the object.

This rule is designed to ensure that the combined object remains a single-root tree, or in other words, a hierarchical object that Gneiss recognizes. This allows the joined object to be used in Gneiss the same way as other regular hierarchical objects in the spreadsheet. After the joined object is created, Gneiss's visualization algorithm creates the view of the joined object based on the order of the fields in spreadsheet columns. The joined object can then be reshaped through drag-and-drop, regrouped by selected fields, and sorted and filtered by its structure. In

conventional spreadsheets, to join two tables, the user has to either use `LOOKUP` functions to connect the joined columns, or write SQL queries to join the tables. Some research tools such as [23] flattens hierarchical objects into tables and joined them using SQL techniques. However, as mentioned before, both conventional spreadsheet mechanisms and SQL work on flat table data and cannot support what Gneiss supports for manipulating hierarchical data. The differences between Gneiss's approach for joining hierarchical data and conventional approaches is further discussed in the related work chapter in section 2.4.

Also, as illustrated in Figure 5., Gneiss's rule for joining may discard nodes in the non-preserved tree if there is no matching node in the preserved tree (such as the node "2" in the red tree in Figure 5.). One can consider Gneiss's joining similar to SQL's `LEFT JOIN` or `RIGHT JOIN` where one table is preserved and the other table is connected to the preserved table by the joined fields (although in Gneiss, the preserved object is decided by the system for reasons discussed above). Currently, Gneiss's joining algorithm does not support preserving only the common joined fields (similar to SQL's `INNER JOIN` command) or preserving all joined fields (similar to SQL's `FULL OUTER JOIN` command). One can see that preserving only the common joined fields to be an easy extension to the current system (for example, in Figure 5.11, to just remove the node 1 from the joined tree). Supporting preserving all joined fields, however, is tricky since the algorithm has to decide a node's parent when there is no matched node in the preserved tree. It is still an open question of how that can be implemented when the joining objects are trees instead of tables.

5.4 DEMONSTRATIVE EXAMPLE

Here I described one more example to demonstrate how users can use Gneiss to create hierarchical visualizations and explore the data. In this example, I use the same conference datasets as in the usage scenario (*papers.json* and *sessions.json*).

5.4.1 PAPER GEOGRAPHY VISUALIZATION

The user is interested in finding out the demographics of the papers – where the papers come from. She wants to explore the data by the continents and countries of the institutions of the papers. She first extracts the paper titles, institutions and countries to column A, B and C. While the *papers.json* file does not have the continent information of a paper, the user can compute the continent of a paper using the country field. There are multiple ways to do this in Gneiss. The user can insert a new column next to the country column and manually enter the continent for each country. Another way is that if the user has a list of countries and each country's continent available in the spreadsheet (which can found very easily online, for example, by googling "countries and continents json"), she can use the `LOOKUP` function to find the continent of the first institution's country, and then autofill the

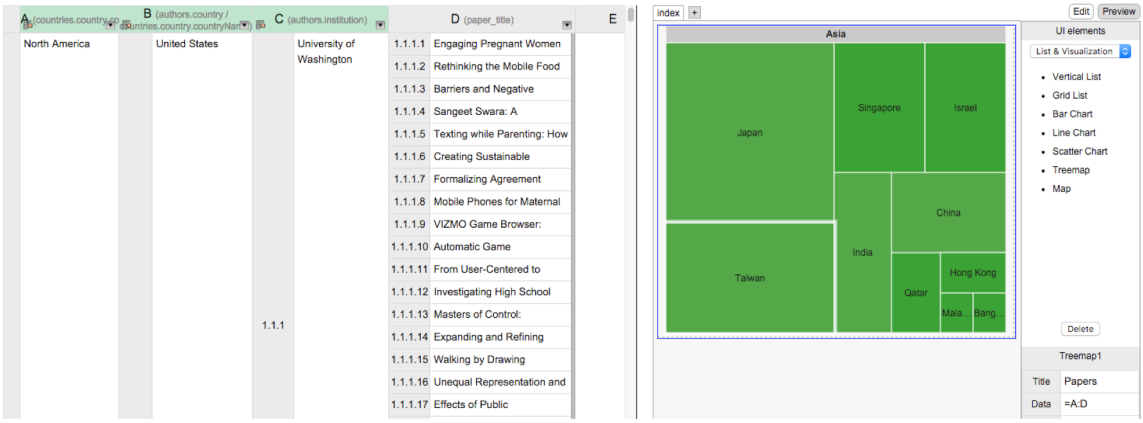


Figure 5.12. A screenshot of this demonstrative example. In the spreadsheet, the user groups the CHI'15 papers by their continents, countries and institutions. The user then visualizes the spreadsheet data using a treemap in the right web interface builder. Through the treemap, the user can view the distribution of data in each hierarchical level. For example, she can tell from the treemap that in Asia, Japan, Taiwan and Singapore are the countries that have the most papers.

rest of the cells in the continent column as described above. Alternatively, as described earlier, Gneiss further provides the ability to join two tables together in a spreadsheet using interaction techniques without using spreadsheet formulas. So as the user acquires a list of countries and their corresponding continents, she can directly join the paper data with the acquired list by the country field to fill in the continent information for each country. After the user has in the continent information for each institution's country, she drags the columns to put the continents in column A, countries in column B, institution names in column C and finally the paper names in column D. She then groups the data by column A, B and C respectively, and removes duplicate paper titles in column D for each institution using filtering.

The user gets the data she wants (Figure 5. in the spreadsheet). However, it is still difficult to tell the distribution of the papers because there are so many of them. To further explore the data, in the right pane, the user drags in a treemap visualization from the sidebar, and sets the "data" property of the treemap to `=A:D`. In Gneiss, the treemap visualization [45] takes a hierarchical object (such as the one in column A to D in Figure 5.) and shows each hierarchical level's data using rectangles whose value is the node's value and size is proportional to the number of children the node has. The user can click on a rectangle in the visualization to go to the next level of hierarchy to view all children of a node, and right click anywhere in the map to go back to the previous level[10]. With the treemap visualization, the user easily can find out the top continents that have the most papers, and the click on a continent to view the top countries in the continent that have the most papers (such as in Figure 12 at the right), and so on. As the spreadsheet can now clearly represents the

[10] As described in section 3.2, Gneiss uses Google Visualization API for all its visualizations. Using left and right clicking to go up and down a level in a treemap is the default behavior provided by the API.

hierarchy of data, generating a hierarchical visualization of the data becomes straightforward.

5.5 USER STUDY

I also conducted a lab study to evaluate whether the various new features described above for supporting hierarchical data in spreadsheets introduced in Gneiss could be understood and used by spreadsheet users, and whether these new elements could help users use hierarchical data more efficiently compared with current tools.

5.5.1 STUDY DESIGN

We evaluated our tool by comparing it to how end-users and professional programmers currently work with hierarchical data. We designed a set of data exploration tasks for the study and measured the success rate and the time participants spent completing each task. Our study uses a between-subject design and has three groups. The experiment group used our tool (Gneiss). For the end-user comparison group, we picked Microsoft Excel, the most popular conventional spreadsheet tool. For the programmer comparison group, we picked JavaScript and Python, as our informal poll showed that those were the most popular languages for using JSON data. Participants in the programmer group picked either one of the languages to use in the study. They used Sublime Text as the editor, and viewed the program output in Sublime's Python console (Python) or Chrome's developer console (JavaScript).

5.5.2 PARTICIPANTS

We recruited 12 spreadsheet users and 6 programmers for the study (ages 21-39). All but 2 participants were university students from different departments including computer science, engineering, psychology, public policy and management. The other 2 participants were alumni. Participants rated their proficiency with Excel, programming in general, and using JSON data on a five-point scale from "none" to "superior". Among the 12 spreadsheet users, four rated their Excel proficiency as "3: Intermediate - know how to use basic functions such as `SUM` and `IF`", and eight rated as "4: Advanced - know how to use pivot tables and advanced function such as `LOOKUP`". None of the spreadsheet users considered themselves as programmers, although some of them had prior experience programming using languages like R or MATLAB. Their average rating on programming proficiency was 2.17. A few of them had used JSON data (average rating 1.42). Thus, we consider them to be intermediate to experienced spreadsheet users who are not experienced programmers. The twelve spreadsheet users were randomly assigned into the Gneiss or Excel group (6 in each group). There were no significant differences in the demographic measures between the two groups.

All 6 programmers were in the programming group. 4 of them chose to use JavaScript and 2 chose Python. Their average self-rating for proficiency was 4.17 (out of 5) on programming in general and 3.67 on the language they used in the study. All of them were familiar with JSON data (average rating 3.83). Thus, we consider them to be experienced programmers.

5.5.3 DATA

We used the same CHI'15 conference data described in the usage scenario (*papers.json* and *sessions.json*). There were 484 papers and 119 sessions in the datasets. Participants in the Gneiss and programming group received the data as two JSON files. For the Excel group, we converted the JSON files into CSV files. To our knowledge, there is no standard on how to convert a JSON file into a spreadsheet. Based on our informal interview and pilot tests with several expert Excel users, we decided to provide each JSON file in both a *wide* table and a *long* table spreadsheet out of fairness as both formats were suggested by the expert users. Thus the Excel group was given four sheets (two sheets for each JSON file, put in the same Excel workbook for ease of use). As described in section 5.1, the main difference between a wide and long table is that in a wide table, a list of values within an item is expanded horizontally as multiple columns. Each item (a paper or a session in our data) is a row in the spreadsheet. However, a row could have many empty cells as the lists for different items could have different lengths (such as different papers may have different numbers of authors). In contrast, in a long table, a list of values is expanded vertically. While the long table has no empty cells and fewer columns, it has a lot of duplicated values as an item could be repeated multiple times depending on the length of the lists in each item. For example, a paper with 2 authors and 3 keywords would become 6 rows as each author and keyword was paired once. Participants in the Excel group were told they could use any of the tables to complete the tasks.

5.5.4 TASKS

We designed 5 tasks for the study. The first task is to report the number of papers that have "social" in their keywords. This is the easiest task. The second task is to find the top three institutions that have the most papers (same as the first task in the scenario of section 5.2). For the first two tasks, participants only needed to use the papers file. They had at most 10 minutes to finish each of these tasks. The third through fifth tasks involve using both the papers and sessions files. The third task was to come up with a schedule that could let a person go to the session that has the most award papers in a time slot (similar to the second task in the scenario). The fourth task is to find all the papers that were presented on Tuesday that are from Carnegie Mellon University. The fifth task is to find all the authors who had multiple

papers scheduled to be presented in the same session (a similar task to finding if anyone has conflicting presentations). For the third to fifth tasks, participants had at most 15 minutes to complete each task. Participants in all three groups received the same five tasks and did the tasks in the same order with the same maximum time per task.

5.5.5 PROCEDURE

The study took about 75 minutes per participant. After signing the consent form, participants in all groups received tutorials on the given tool using a tutorial JSON document different from the study data. The Gneiss group received a 20-minute tutorial on all the features described above that they might need. The Excel group received a short introduction to relevant parts of Excel's interface (including sorting, filtering, the pivot table and the remove-duplicates widget) and a tutorial on the long and wide table formats. The programming group received a short introduction to the Sublime editor and the JSON files. Participants in the Excel and programming groups were given up to 10 minutes to get familiar with the editor they used and to install any libraries or plugins they wanted to use. Two programmers used jQuery, one programmer used D3.js and lodash. None of the Excel participants installed any plugins.

After introducing the tool, the experimenter showed and explained the study data to the participants. Then they began to do the tasks. Participants in the Excel and programming group could look up anything online. Participants using Gneiss were given an A4 paper with a list of the spreadsheet functions supported in the system. In all conditions, when participants thought they were done with a task, the experimenter checked the answer and requested the participant to continue if their answer was incorrect. This checking time was not counted as part of the participants' time. After doing the tasks, participants filled out a short survey on their demographics and feedback (see Appendix C). They were then paid $15.

5.5.6 RESULTS

For each task we measured the task completion time. If the participant did not complete a task within the time limit, we used the time limit as the task completion time. We analyzed each task using a Kruskal-Wallis Test and post hoc analysis. The results are reported below and in Figure 5.13. All time numbers are in seconds.

Task 1: All participants completed this task. There was a significant effect of the average task completion time for the three groups ($Chi^2(2) = 6.428$, $p = .040$). Post hoc comparisons showed that the Gneiss group spent significantly less time ($M = 142.83$, $SD = 119.84$) on the task than both the Excel group ($M = 370$, $SD = 142.23$; p

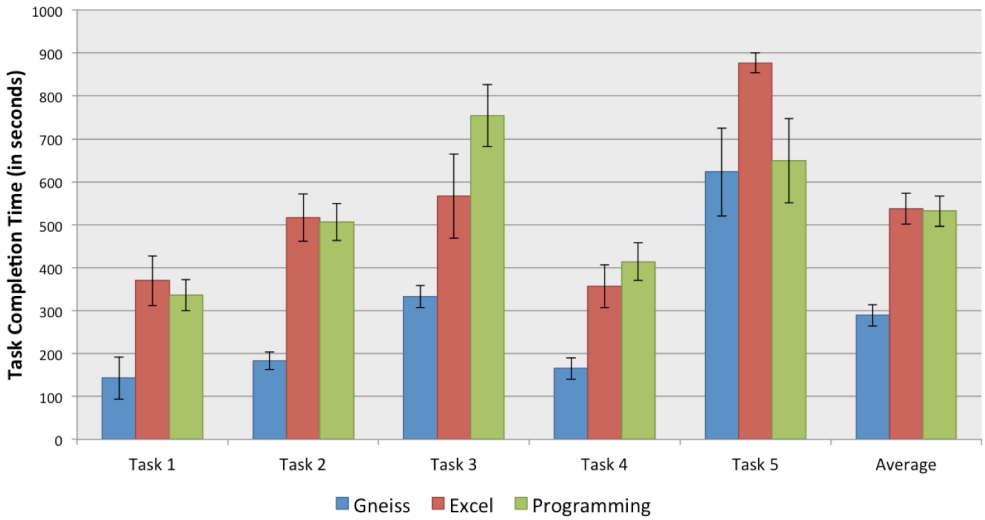


Figure 5.13. The average task completion time for the Gneiss, Excel and programming groups. Shorter bars are better.

= .037) and the programming group (M = 336.17, SD = 36.66; p = .025). The difference between the Excel group and the programming group was not significant.

Task 2: All participants using Gneiss completed the task. Three participants in the Excel group and one participant in the programming group did not complete the task within the time limit. There was a significant effect of the average task completion time for the three groups ($Chi^2(2) = 11.986$, p = .002). Post hoc comparisons showed that the Gneiss group spent significantly less time (M = 183.5, SD = 50.35) on the task than both the Excel group (M = 517.17, SD = 135.58; p = .004) and the programming group (M = 506.5, SD = 107.20; p = .004). The difference between the Excel group and the programming group was not significant.

Task 3: All participants using Gneiss completed the task. One participant in the Excel group and one participant in the programming group did not complete the task within the time limit. There was a significant effect of the average task completion time for the three groups ($Chi^2(2) = 10.551$, p = .005). Post hoc comparisons showed that the Gneiss group spent significantly less time (M = 183.5, SD = 50.35) on the task than both the Excel group (M = 517.17, SD = 135.58; p = .025) and the programming group (M = 506.5, SD = 107.20; p = .004). The difference between the Excel group and the programming group was not significant.

Task 4: All participants completed the task. Again, there was a significant effect of the average task completion time for the three groups ($Chi^2(2) = 9.906$, p = .007). Post hoc comparisons showed that the Gneiss group spent significantly less time (M = 165.67, SD = 62.42) on the task than both the Excel (M = 357, SD = 123.67; p =

.016) and the programming group (M = 414.17, SD = 107.53; p = .004). The difference between the Excel group and the programming group was not significant.

Task 5: Task 5 is the most difficult task. Five participants in the Excel group, two participants in the Gneiss group and two participants in the programming group did not complete the task within the time limit. Kruskal-Wallis Test showed that there was no significant difference in the task completion time among all three groups.

Averaging the five tasks: There was a significant effect of the average task completion time for all five tasks for the three groups ($Chi^2(2) = 11.380$, p = .003). Post hoc comparisons showed that the Gneiss group spent significantly less time (M = 289.37, SD = 61.58) on the tasks than both the Excel (M = 537.43, SD = 89.43; p = .004) and the programming group (M = 531.93, SD = 87.04; p = .004). The difference between the Excel group and the programming group was not significant.

In summary, our results showed that overall, participants using Gneiss completed the tasks almost twice as fast as participants using Excel or programming.

Subjective Results

I asked participants in a post study survey about their feedback on the tasks and the tool they used. Using a 7-point scale, participants rated the tasks highly realistic (average rating Gneiss = 7, Excel = 6.16, programming = 6.5, overall = 6.56) and close to what they do in real life (average rating Gneiss = 4.67, Excel = 4.33, programming = 5, overall = 4.67). I also asked the participants how easy they thought the tasks were. The programming group rated the tasks the easiest (average rating 3.25), followed by the Gneiss group (average rating 3.67). The Excel group rated the tasks to be the hardest (average rating 4.83). This is interesting since the programming group did not perform the best (although the differences among groups are not statistically significant). I discuss this result further in section 5.6.4. Gneiss participants rated the tool easy to learn (M = 5.67) and could see themselves using it in their own work (M = 5.83).

5.6 DISCUSSION

5.6.1 LEARNING AND USING GNEISS

All six participants using Gneiss successfully used its novel features to complete the tasks after receiving a 20-minute tutorial. Some participants took a longer time than others to understand the meaning of the nested tables. Although we explained that the nested tables are a way to show hierarchies in data, some participants viewed it more as a way to show one-to-many relationships and were a little confused at first if they saw a nested table having only one row. This, however, did not affect the participants in solving the tasks. We found that even when the participants did not

fully master the visualization rules, they could still successfully move the columns into the correct places eventually by trying out different combinations by dragging a column to different locations. This low-cost method of reshaping the data facilitated opportunistic data explorations in the Gneiss group and often led the participants to the correct answer. As the study went on, all participants gradually became more familiar with the visualization rules, and they seemed to act more quickly and needed fewer trials.

In addition to the features on using hierarchical data, the study also tests the learnability and usability of the source pane. None of the Gneiss participants seemed to have any trouble using it to view and extract the appropriate JSON data. While none of the Gneiss participants were familiar with JSON syntax, they were able to understand the study data very quickly and use the drag-and-drop gesture to extract the desired fields from the source pane to the spreadsheet. This eases some concerns I originally had that end-users may be intimidated by the JSON data syntax and may not know how to get started. One reason that participants were able to use the JSON data in the study easily could be because they all had some prior knowledge about the data – all participants were university students familiar with what attributes a conference paper may have, such as keywords and authors. The result suggests that a JSON document, if formatted and structured properly, could be understood by end-users with domain knowledge related to the data, even if the users are not familiar with the JSON syntax.

5.6.2 STRENGTHS OF HAVING HIERARCHIES IN SPREADSHEETS

As we expected, participants using Excel were constantly troubled by the repetitive data in the spreadsheet caused by flattening a hierarchical document into a table. The Excel participants chose to use the long table format in most tasks as the wide table contains too many columns and thus is difficult to read. The key to using a long table is to remove unneeded duplicate values. Many of our tasks required the user to remove duplicate rows using *multiple* columns. For example, in the second task, the participants had to get a list of unique paper title and institution pairs before using pivot tables to calculate summaries. We found that while removing duplicated values in a single column was intuitive to the participants, the concept of removing duplicates using multiple columns (or in other words, to create unique keys using multiple attribute values) was difficult for the spreadsheet users to figure out how to do.

Gneiss successfully avoided this problem by supporting hierarchical grouping and filtering. For example, in Figure 4, Gneiss users could interactively merge data first by a column and then by another column to get rid of duplicate rows using multiple columns. There was visual feedback at each step that led the users naturally to the next operation (e.g., in Figure 4 at 4, it was intuitive to group the data again by

column B after seeing the repetitive papers within an institution). However, in Excel, since there were no hierarchical representations in the spreadsheet, removing duplicate rows by multiple columns cannot be done in sequence. Excel does provide a "remove duplicates" widget, but to remove duplicate rows by multiple columns using that widget, the user either had to make a new column that uses values from both of these columns (to create a key) and remove duplicates using that new column, or had to know to have multiple columns checked at the same time in the remove duplicates widget. Neither of these strategies was straightforward for the participants based on our observations.

We also observed advantages in using Gneiss' grouping and spreadsheet functions to compute summaries of data instead of using pivot tables. The biggest advantage is that participants with Gneiss could see the data while calculating the summaries and thus were more likely to spot problems in their manipulation of the data, such as duplicated values. In contrast, in Excel, as the pivot table interface took people away from the original data, it was more difficult for participants to spot errors.

5.6.3 STRENGTHS OF HAVING A VISUAL TOOL TO WORK WITH DATA

We observed that all participants in the programming group constantly printed out data to the console to check if their code manipulated the data in the way they wanted. Participants in the programming group also gave more incorrect answers before having the right answers than participants using Gneiss and Excel, since they did not have a visual way to examine the manipulated data and thus were more likely to miss errors in their code like omitting to deal with repetitive values. This result suggests to us that even professional programmers might benefit from having a visual tool (such as spreadsheets) to work with data, especially when doing data exploration tasks.

5.6.4 LACK OF COMPUTATIONAL THINKING

The fifth task was the most difficult task for spreadsheet users because solving it requires computational thinking. Instead of getting rid of duplicated values, in this task the user needs to discover a way to keep authors that have papers with duplicated session ids and get rid of the ones that do not. One strategy is to create another column that computes whether there were duplicated values. In Gneiss, this can be done by checking if the `COUNT` and `COUNTUNIQUE` functions return different values. This approach, which may be obvious to a programmer, is not straightforward to spreadsheet users who do not have much programming background. The two Gneiss users who failed this task both had organized and grouped the data into the right form but could not think of a way to describe this relationship. Another interesting observation was that while programmers on average spent longer solving the tasks than Gneiss users, they rated the tasks easier

than either the Gneiss or Excel users. While statistically the differences are not significant, this result makes sense, as although it took longer for programmers to write a working program, the logic for solving these tasks was pretty straightforward. This result suggests that while our tool extends spreadsheets to support many new ways to manipulate hierarchical data, whether users could successfully use our tool to solve a task in real life still may be limited by their programing training.

5.6.5 POSSIBLE FORMS OF OUTPUTS

We showed Gneiss to participants from all groups after the study was over. Participants expressed interest in using it in real life. Besides using it as a data exploration tool, they also suggested some possible forms of outputs that could be useful to them. One programmer suggested that Gneiss should be able to export a hierarchical table into a new JSON file that he could then feed into his programs, which would be trivial to provide, since Gneiss creates such a table internally. Another programmer wanted to use Gneiss as a programming-by-demonstration tool to generate data reshaping scripts, such as a snippet of JavaScript code for him to paste into his program. Another programmer wanted Gneiss to become a database console that can generate queries to databases using spreadsheet interactions like sorting, filtering, grouping and joining. Several programmers and spreadsheet users wanted Gneiss to support creating hierarchical visualizations, such as treemaps or sunburst graphs. After the study, I have added several hierarchical visualizations to Gneiss's web interface builder including treemaps and nested lists. The rest of the suggestions could be interesting directions for future work.

5.7 LIMITATIONS

There are limitations in Gneiss's system on handling hierarchical data and of the user study. For the system, as I discussed earlier in section 5.3.1.1, Gneiss currently supports JSON data that use arrays to represent hierarchies. However, different data formats may have different ways to specify hierarchies. For example, XML does not have arrays and the hierarchies are specified using nested tags. While Gneiss currently does not support other hierarchical formats, I believe that Gneiss's techniques can be adapted to handle those data as well. Also, as I mentioned in section 5.3.2.6, Gneiss's joining algorithm does not let users choose which object to preserve, and nodes in the non-preserved object are discarded if the preserved object do not have matching joined fields. To my knowledge, currently there is no standard on how to join hierarchical objects. Gneiss's joining algorithm is designed to work together with the visualization algorithm and with other techniques for manipulating hierarchical data. Finally, currently Gneiss does not allow users to create arbitrary data structures – users can only create a new column that has the

same structure as a selected column, which comes from the source hierarchical document. Since my user study and the study in [7] both showed that people could understand nested cells in spreadsheets, future work could experiment with using spreadsheets to create arbitrary hierarchical data.

As for the user study, while the results were statistically significant, they were collected from a small group of participants that were mostly university students. Therefore the results may not be generalizable to users with different demographics. The purpose of the study was to evaluate the learnability and usability of Gneiss and provide some insights on the strengths and weaknesses of Gneiss's design.

5.8 CONCLUSIONS

In this chapter, I presented a spreadsheet model for using hierarchical data. The model supports reshaping, regrouping and joining hierarchical data in a spreadsheet using simple interaction techniques. Conventional spreadsheet mechanisms including spreadsheet languages, sorting, filtering and autofilling are extended to further support manipulating data using hierarchical structures. To evaluate this model, I presented a series of examples to demonstrate the ability of the model to support different data analysis tasks. I also conducted a lab study where Gneiss helped spreadsheet users with little programming experience complete realistic tasks significantly faster than Excel and even outperform professional programmers writing code, showing that hierarchical representations can be successfully integrated into spreadsheet-like tools.

CHAPTER 6 USING STREAMING DATA[11]

Many web data services return real-time data such as market prices, geo-locations of people and vehicles, or social network feeds. Applications that use real-time data sources often need to analyze the data live to provide feedback. For example, an application that monitors the stock market might want to give the user a notification if the buying price is below a certain number. Another common usage is to periodically stream data from real-time sources in order to analyze the data over time. For example, a business owner may want the data on the number of daily Twitter feeds that contain a hash tag of the company's name over the past month to see if it is correlated with the dates of the company's sales.

6.1 MOTIVATION, CHALLENGES AND CONTRIBUTIONS

Currently, creating custom applications that use real-time data sources requires a programmer to write a lot of code to collect the data, manipulate it, and analyze it live to update the user interface. Often such applications would require a database to store the retrieved history data and support querying the data over time such as to get data collected within a certain time period. This requires additional programming efforts. For end-users, there are tools that provide built-in real-time data sources to let users view and analyze live data using spreadsheets and visualizations (e.g., [83,98,115]). However, adding a new data source to use in those tools requires a programmer to hardwire it into the tool. Also, none of those tools support programming GUI applications that use streaming data sources.

In this chapter, I describe a spreadsheet model integrated with Gneiss that lets users work with streaming data from web data sources. The model has several innovations: first, it provides techniques that allow users to stream any fields from arbitrary REST JSON web services without needing a developer to preprogram those sources into the tool. Second, it introduces a design for spreadsheet cell "metadata" which describes other attributes of a cell's value and can be used to manipulate spreadsheet data. In this work, each cell automatically records metadata of its value's provenance and fetched time, allowing users to view or manipulate streaming data in the spreadsheet using temporal information, such as getting the daily maximum and minimum values. Lastly, it allows streaming to be paused and restarted using conditions computed live from spreadsheet data using formulas. These features make the created spreadsheet program very dynamic and interactive.

To show the generalizability of this model, I demonstrate how the same mechanism that handles streaming data from web services can be used for collecting user data

[11] The research in this chapter is also described in our publication at CHI'15 [21]

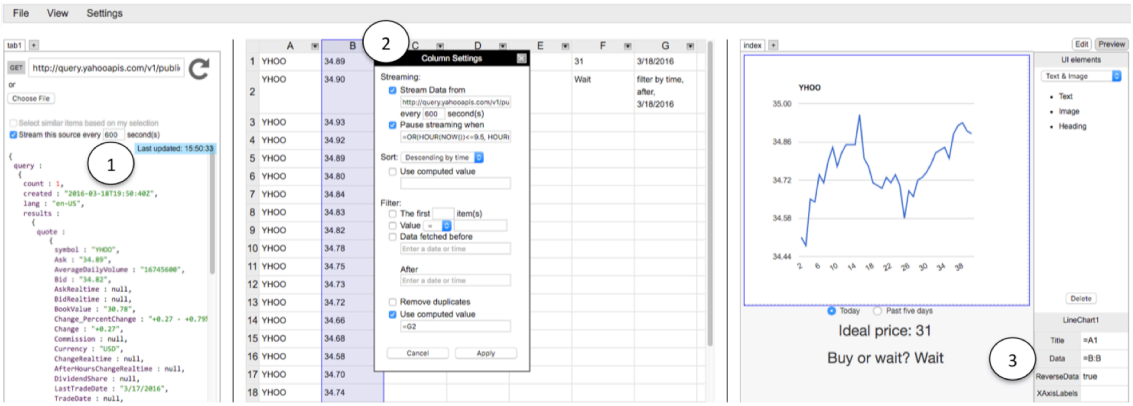


Figure 6.1. The user is creating a real-time web application (the right pane) that streams Yahoo's stock prices (columns A and B in the spreadsheet) every 10 minutes. To let Gneiss periodically pull data from a web service, the user selects the "Stream this source" checkbox (1) in the source pane. The system shows the source pane data's last retrieved time in a blue label. Gneiss's spreadsheet allows users to set custom streaming frequency and pausing rules, and to sort and filter the data by fetched time (2). In this screenshot, the user uses a line chart (3) to see the ups and downs of today's prices. Full description of this example screenshot is in section 6.2.

in web input elements such as textboxes on web pages. A spreadsheet column can be set to pull data from a web UI element either when the input element changes or when triggered by live conditions. The data will be saved as a data stream in the spreadsheet, making the spreadsheet work as a backend database for the web application.

Combining all these features, my dissertation contributes a novel spreadsheet model for using streaming data, where custom real-time applications need only a few spreadsheet formulas that otherwise would require writing complex code.

6.2 USAGE SCENARIO

Like in previous chapters, here I describe a usage scenario to give an overview on how Gneiss supports creating a GUI web application that uses real-time data. In this scenario, the user is making a web application that streams the stock price of Yahoo every 10 minutes. The streaming automatically pauses after 4PM when the market closes, and resumes after 9:30AM when the market opens again. The web application can let users choose to view the data of today or the past three days using two radio buttons. The data are shown in a line chart. Below the line chart, the application shows if the current price is below a value that the user sets previously. This example is similar to the example described in [83], where the user also creates a spreadsheet that shows streaming data of market prices and uses spreadsheet formulas to compute if the current price is a bargain. In my example here, the user can further define custom pausing rules, select the data using temporal information and create a web application that uses and manipulates streaming data in the spreadsheet. To my knowledge, none of the prior spreadsheet systems support those features, including [83].

To begin, the user enters Yahoo Finance API in the URL bar in the source pane to get the current market price of Yahoo. Then he checks the "Stream this source every *[textbox]* seconds" checkbox (Figure 6.1 at 1), and uses the textbox to set the streaming frequency to be 600 seconds (Figure 6.1 at 1). The system starts to pull data from the data source every 10 minutes, and uses a "flipping" animation in the source pane to show that data has been refreshed. To stream the value of the ask price to the spreadsheet, the user selects the "Ask" field in the return data, and drag-and-drops it to spreadsheet column B. By default, streaming data in a column are sorted descending by time. So the system starts to stack column B with the latest "Ask" value retrieved from the web service, with the newest value appears at the top. For clarity, the user also extracts the name of the company ("YHOO") to column A. Data in column A and B now grow live as time goes by, adding a new row every 10 minutes.

The user drags a line chart to the right pane and set the "Data" field of the chart to be `=B:B` (Figure 6.1 at 3). The chart now plots Yahoo's stock price and updates every 10 minutes with a new data point. The user then drags two radio buttons below the chart to control whether to show the data of today or the past five days. In cell G1, the user enters the formula

```
=IF(RadioButton1!Checked, TODAY(), TODAY()-5)
```

`TODAY` is a conventional function supported in spreadsheets such as Excel that returns today's date and automatically updates every day. The user then presses the arrow button at the top of column B to open the dialog box for sorting and filtering. Recall that in section 4.3.2, I had described that sorting and filtering rules in Gneiss can be computed dynamically from GUI elements in the web application. To do so, the user selects the "Use computed value" checkbox for sorting or filtering, and in the textbox below enters the computed value which is a string that has the sorting or filtering method along with the required parameters of the method (see **Error! Reference source not found.** for reference). Here, the user wants to filter the data to only show data retrieved on and after the date that is shown in cell C1. To do so, in cell C2 the user enters `="filter by time, after, "&C1` to compose the string for filtering. The user then selects "Use computed value" in the dialog box for filtering, and enter `=C2` in the textbox (see the dialog box at Figure 6.1 at 2). This makes the system only show data retrieved on and after the date showed in cell C1, which is controlled dynamically by the radio buttons in the web interface builder. Since the two radio buttons are in the same radio button group, selecting one button automatically deselects the other. Therefore the `IF` function only needs to check `RadioButton1`.

The user now wants to let the system only stream data when the market is open, which is between 9:30AM to 4PM. To do so, he selects the "Pause streaming when" checkbox in the dialog box and enters `=OR(HOUR(NOW())<=9.5, HOUR(NOW())>=16)`. `NOW` is another conventional spreadsheet function that returns the current date and time. In Gneiss, the `NOW` function updates every minute. This expression makes the spreadsheet pause streaming when the current time is before 9:30AM or after 4PM. He clicks "OK" to apply these custom rules and close the dialog box.

Finally, the user wants to compare the latest market price with a predefined price. He enters the formula `=IF(B1>F1, "Buy", "Wait")` in cell F2. The formula compares the latest value (which is stored in B1 as the data is sorted descending by time) with the "ideal price" in cell F1 (see Figure 6.1 in the spreadsheet) and dynamically outputs "Buy" or "Wait". In the web interface builder, the user drags in a few text labels to show the values of F1 and F2.

The user now has finished this web application! The user sets the system to keep streaming the data even when Gneiss is closed using a menu option under "Settings" in the top menu bar (Figure 6.1 at the top), and exports the application. Now he can open this application in any device that has a web browser. The chart and the "buy or wait" text in the application will both update live, and the line chart data are controlled by the radio buttons.

6.3 KEY FEATURES FOR USING STREAMING DATA

As described in this chapter's usage scenario (section 6.2), the user can let the system periodically pull data from a web service by checking the "stream this source" checkbox in the source pane, and then create a data stream of a field in a spreadsheet column using the drag-and-drop gesture. As described in previous chapters, to sort and filter data by a column, the user clicks on the arrow icon at the top of a column to open a dialog box. If the column holds streaming data, the dialog box will have a new section at the top called "Streaming:" (see Figure 6.1 at 2) that shows the web API that the data is streamed from and a checkbox for pausing streaming. The dialog box also provides new options to sort and filter the data using temporal information, as now the data in a column are retrieved at different times.

6.3.1 SPREADSHEET CELL METADATA

Streaming data are inherently time-series data, and so the ability to view or manipulate streaming data in the spreadsheet by time is essential. To enable this, I designed each spreadsheet cell to have metadata that describe attributes of its value. The metadata are by default not visible but can be exposed through formulas and can be used to manipulate, sort and filter spreadsheet cells (note that cell metadata in Gneiss are different from user comments in conventional spreadsheets

as they are set and maintained automatically by the system). In my spreadsheet model, each streamed cell stores not only its display value but also metadata about its provenance and fetched time, allowing data to be viewed and manipulated using its value, source and temporal information.

6.3.1.1 Sorting and filtering the data by fetched time

As described in the usage scenario, a data stream in a spreadsheet column can be sorted and filtered using fetched time. By default, data in the spreadsheet is sorted descending by time, so that the latest value appears in the first row. The user can also sort the data ascending by time and let the latest value be added to the last row. For filtering, the user can filter the data to show values fetched before and/or after a certain time. To do so, the user selects the "Data fetched before *[textbox]* after *[textbox]*" option, and enters the before/after time in the textboxes (see Figure 6.2). I used Datejs, a JavaScript library that parses natural language into standard date and time format, to enable more flexible date entry formats. As the user types a time into the textboxes, the system will show the parsed time below the textbox in small grey text (Figure 6.2). The user can check if the system interpreted the time correctly, and if not she can include more details.

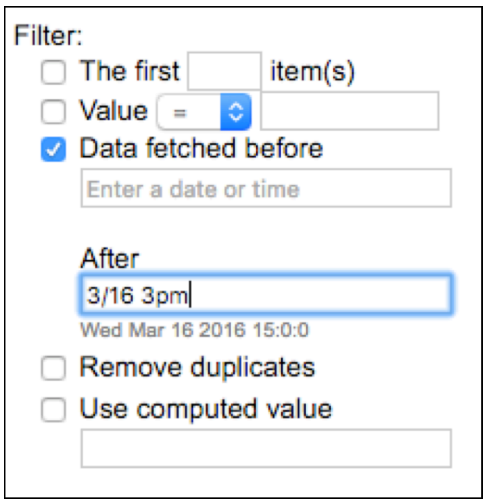


Figure 6.2. When entering a time string to filter the data, Gneiss allows flexible formats, parses the user's input and shows the system interpreted time in grey text below the textbox.

As described previously in sections 3.4.3 and 4.3.2.2, sorting and filtering rules can also be dynamically computed from spreadsheet cells and web GUI elements using formulas. I have provided multiple examples in previous chapters, and the usage scenario in this chapter (section 6.2) gave another example of how the user can compose a string whose value is controlled by radio buttons on the web application to filter the data by time. To facilitate getting the current time, Gneiss extends two conventional spreadsheet functions, `TODAY()` and `NOW()`. Both functions automatically update in real-time, with `TODAY()` updating daily and `NOW()` updating every minute.

Again, sorting and filtering on streaming data are re-evaluated when new values arrive or when the rules change (if they are dynamically computed from spreadsheet cells or functions). Adjacent spreadsheet columns extracted from the same streaming source are sorted and filtered together (see section 3.4.3), and they are highlighted in the same color when the dialog box is open.

6.3.1.2 Using cell metadata in spreadsheet functions

Gneiss further supports two new functions to enable users to use cell metadata in formulas in the spreadsheet. The function `FETCHTIME(cell)` returns the retrieval time of a streamed cell. The return value is in standard ISO 8601 format and can be used with conventional spreadsheet time functions such as `HOUR` and `DATE`. The other function `SELECTBYTIME(range, startTime, endTime)` returns values in `range` that are streamed between `startTime` and `endTime`. The function can be used together with many conventional spreadsheet functions that process a list of values. For example, suppose column B in the spreadsheet holds latest news streamed from a news data source. The formula:

```
=COUNTIF(SELECTBYTIME(B:B, "2016-02-29 9:00", "2016-02-29
                10:00"),"*White House*")
```

returns the number of news articles streamed before 9-10am on February 29th, 2016 that contain the phrase "White House".

6.3.2 CONTROLLING STREAMING TIMING

My spreadsheet model also allows the user to set when and how fast the spreadsheet should pull data from a data source. In Gneiss, the user can set the streaming frequency in two places. First is from the source pane using the textbox in the "Stream this source" checkbox option (Figure 6.1 at 1) as described in the usage scenario in section 6.2. Second is from the dialog box of a streaming column (Figure 6.1 at 2). The frequency data in these two places are synchronized – changing the number in one place will automatically change the number in the other.

In the dialog box, the user can also choose to pause a stream when a given condition is true. For example, in the usage scenario, the user sets the system to only stream data between 9:30AM and 4PM using a `NOW` function that checks the current time every minute. If the condition is not specified, the stream pauses immediately when the user checks the "Pause streaming when" checkbox.

By default, all streaming stops when the spreadsheet or the created web application are closed. As described in this chapter's usage scenario, Gneiss also provides the option to keep streaming when the spreadsheet or the created web application was closed (although our server has a limit for how much data can be stored in the database[12]), or to remove all the streaming data saved on the server. When the user opens the spreadsheet or the created web application, Gneiss fetches the data stored

[12] We use MongoDB as the database. See its limitations at http://docs.mongodb.org/manual/reference/limits/

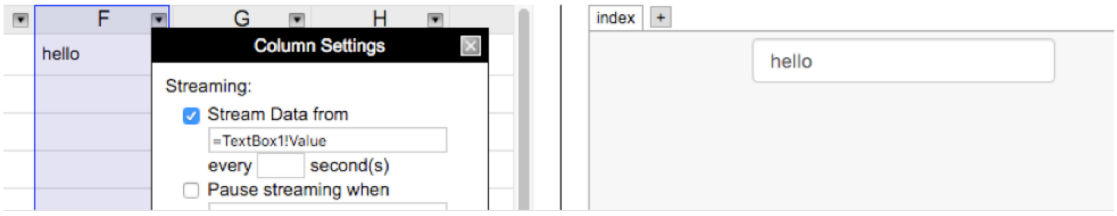


Figure 6.3. The user can also stream data from web input elements to a spreadsheet column, by having the streaming source to be a GUI element property.

on the server based on the sorting and filtering rules to fill in the spreadsheet cells and the web application, and restart all the streaming.

6.3.3 STREAM DATA FROM WEB INPUT ELEMENTS

I applied this spreadsheet model for streaming data from web services to streaming data from web input elements in a Gneiss web application. As streaming data are stored on Gneiss' server, enabling data to be streamed from web input elements essentially turns a spreadsheet into a database for a web application that stores user inputs. To stream and store input values in the web application to the spreadsheet, the user first enters a web GUI element property, such as `=TextBox1!Value`, in the first row of a column. Then when she opens the dialog box of that column, the "Streaming:" section will appear with the streaming source textbox being what was entered in the first cell (see Figure 6.3), as the system detects that the first cell is a web GUI element which can be a streaming source. Then the user can set the column to pull data from the web element by checking the "Stream data from" checkbox. By default, the column pulls data from an input element whenever its value changes. The textbox's value attribute by default changes when the user presses the enter key in the textbox, and that is when the system pulls the data to the spreadsheet column. The user can further use the "pause" mechanism described earlier to *start* streaming only when certain condition is true. For example, using `Button1!State != "clicked"` as the pause condition makes the column pull data from `TextBox1` only when `Button1` is clicked (see the second demonstrative example in section 6.4.2 for a use case). Like spreadsheet cells storing data streamed from web services, cells storing data streamed from web input elements also have the same metadata and can be manipulated by their retrieval time.

6.4 DEMONSTRATIVE EXAMPLES

As in previous chapters, here I describe two more examples to demonstrate Gneiss' ability to create real-time applications that stream data from web services and web applications using a few lines of spreadsheet code.

6.4.1 REAL-TIME WEATHER ALERTS BASED ON LOCATIONS

Suppose a user works in a company that has an internal web service tracking current locations of company trucks and wants to create an application that monitors the weather condition at a truck's current location to alert the driver of issues such as if the atmosphere visibility becomes too low. The user first sets column A to stream GPS coordinates of the truck from the company web service and to sort the data descending by time (the default), thus the most recent coordinates are in cell A1. To retrieve weather data of the truck's current location, the user uses Yahoo's Weather API, replacing the query value to refer to cell A1 and streams the visibility field from the return data to column B. Cell B1 thus becomes the atmosphere visibility reading of the truck's current location. The user can then use an `IF` formula to check B1's value and see if it is below a certain threshold.

The user can pause a data stream programmatically using live data in the spreadsheet. Suppose the user stores the destination for the truck in cell C1. She can then set the pause condition to be `A1=C1`, so the streaming stops when the truck arrives at its destination.

6.4.2 ONLINE EXAM PAGE

Suppose the user is a teacher who wants to create a test for her students to take online. The test has three pages. On the first page, the student enters her student ID and clicks a "Next" button to go to the next page. The second page shows the student's ID entered in the previous page, a list of exam questions for which the student can enter answers using various GUI controls, and a "Submit" button that submits all the answers and goes to the next page. The third page tells the students that he has finished the exam and shows his score and the class average score so far. If the system finds a student already took the test in the first page, it will skip the second page and take the student directly to the third page showing the message and the scores. To create a custom web application like this currently requires a person to set up a server and a database, and connect the frontend web pages to the server by writing JavaScript, PHP or other web programming code. With Gneiss, the user can create this application using only spreadsheet languages.

The user creates three pages in the web interface builder, naming them index (the first page), questions (the second page) and results (the third page). To the first page, the user drags in a textbox (whose ID is `TextBox1`) for entering the student ID, a "Next" button for going to the second page, and several text labels to explain how to proceed. To check if the student already took the test, the user writes this formula in spreadsheet cell A1:

```
=IF(COUNTIF(TextBox1!Value, B:B)=0, "questions", "results")
```

This formula returns “questions”, the second page’s name, if the value in the ID textbox does not exist in column B, which is where the user plans to store all the IDs of students who already took the test. Otherwise the formula returns “results” whish is the third page’s name. Then the user sets the “Link” property of the Next button to be `=A1` to let it link to the proper next page.

On the second page, the user creates a bunch of text labels for showing the questions and GUI controls for recording the answers. In the spreadsheet, she opens column B’s dialog box and sets column B to pull data from the student ID textbox by entering `TextBox1!Value` as the streaming source and `Button2!State!="pressed"` as the pausing rule to only pull the data to column B when the submit button on the second page (whose ID is `Button2`) is pressed. The user uses other spreadsheet columns to stream data from other GUI elements to save the student’s answer when the submit button is hit using the same method. Finally, she sets the “Link” property of the Submit button to be “results” to go to the third page.

Now on the spreadsheet, each student’s ID and answers are recorded in a spreadsheet row. The great thing about storing everything in a spreadsheet is that the user can now use the familiar spreadsheet functions to check if the answers are right and calculate a final score for the student. Suppose the student’s answers are recorded in columns C to J. The user can then use a bunch of `IF` functions to calculate a student’s final score in cell K1 and then use autofill to compute all students’ scores in column K. On the “results” page, the user drags in a few text labels to tell the student that he has finished the test, and uses the formula `=LOOKUP(TextBox1!Value, B:B, K:K)` to find a student’s final score given the ID (which is still stored in `TextBox1` in the first page). The average score of all students can be easily calculated by the formula `=AVERAGE(K:K)` and assigned to a text label in the web application to show to the student. The user does a few tests of her exam web application, cleans the test data stored in the database before finally exporting the application and sending a link to her students to take the exam. While the application is being deployed, the user can reopen the spreadsheet in Gneiss anytime to look at who already took the test, run more analyses or create visualizations using the collected data.

6.5 LIMITATIONS AND DISCUSSION

In my current implementation, cell metadata and the ability to manipulate data by fetched time only work for cells that have web service data or values streamed from web GUI elements. Cells that have non-streaming or non-web service data do not record the data’s creation time. For example, if the user types a number `10` in a cell, the cell does not record when the number is entered. Future work could be to let cells that have non-streaming data also record their data’s creation time and provenance (if from external sources) to enable other uses. For example, if all cells

record their data's creation time, the user may be able to filter to see cells that changed most recently. In this dissertation, cell metadata focuses on supporting streaming data. Currently, in Gneiss the only way to view the metadata (fetched time) of a cell is through spreadsheet functions. Other future work can be providing other ways to view a cell's metadata, such as through hovering.

Some web services provide streaming APIs that automatically stream data to the client without requiring the client to periodically pull new data, such as Twitter's streaming APIs to get the latest Twitter feeds. Gneiss's source pane currently does not support those streaming APIs, primarily because all of these APIs require using additional authentication protocols, such as OAuth. Besides hard-wiring those APIs into Gneiss, Chapter 3 shows how future work could include integrating Gneiss with the Spinel architecture [17] to enable more complicated data sources to be added to Gneiss as plugins without having to edit Gneiss's source code. After adding a streaming API to use in Gneiss, I expect all the features on manipulating data by time and controlling the timing of streaming could be extended to work with data coming from that source with little modification.

While currently not many web services provide streaming APIs, there are two advantages of using a streaming API instead of periodically calling a regular API as Gneiss does. First, some regular APIs have usage limits that restrict the number of calls an account can make in a time period. The user needs to be aware of this information when setting the streaming frequency in Gneiss. Second, periodically pulling data from a data source does not guarantee capturing all the changes in the data. For example, in the usage scenario, while the spreadsheet pulls the market price every 10 minutes, it cannot capture every buying time where the price is lower than the user-defined value if the price goes up again before the next time the system pulls the data. Capturing that change may be possible if the web service provides a streaming API that pushes data to the client every time when the data change.

A question for systems that deal with streaming data is always scalability. As described earlier, in my current implementation there is a limit on how much data can be stored on Gneiss's server. Most conventional spreadsheet systems including Excel also have a limit on the maximum spreadsheet size, and often the speed of the spreadsheets depends a lot on the machine's memory and system resources. Gneiss uses a client-server architecture (described in the next chapter) and only sends the necessary data to the client. However, it is still possible that the server may have to send a large amount of the data to the client, such as when the user does not set any filtering rules on the data. Some spreadsheet systems for streaming data require the user to set a "window" to limit the amount of streaming data used in the spreadsheet (e.g., [83]). Similarly, Gneiss lets users filter to show only the first X values, although this rule is not mandatory. Making it a requirement that the user

sets a window for streaming data could also be an approach to help solve the scalability problem. My dissertation makes contributions on extending the spreadsheet interface and interaction techniques to support analyzing streaming data and creating live applications that use streaming data. Scalability and performance are not my focus but can be addressed by these known methods.

6.6 CONCLUSIONS

This chapter contributes a model for using streaming data in spreadsheets. It includes techniques to let users stream data from web services and web input elements to a spreadsheet without writing conventional code, a design for spreadsheet cell metadata to let users manipulate spreadsheet data using temporal information, and ways to dynamically control when to pull new data using the spreadsheet language and interaction techniques. Based on this model, Gneiss provides a live environment for analyzing live streaming data and creating database applications that use streaming data sources.

CHAPTER 7 IMPLEMENTATION

In this chapter, I describe the key components of the implementation of Gneiss. Gneiss is implemented as a web application. It uses a client-server architecture where the client handles the user's interaction with the editor and the server deals with data sources and hosts web applications created in Gneiss. Gneiss uses a constraint library for creating the live programming environment. Finally, Gneiss has an internal data model that allows it to support the use of structured data in spreadsheets and web application.

7.1 CLIENT-SERVER ARCHITECTURE

Gneiss is implemented as a web application and uses a client-server architecture. The client side is the Gneiss editor described in the previous chapters that is implemented with HTML, CSS and JavaScript. To create a live programing environment, various components in the editor such as spreadsheet cells and GUI elements in web interface builder are implemented as one-way constraint objects using a constraint library called ConstraintJS [70]. The server is implemented in JavaScript using Node.js as the web server and MongoDB as the database. The communications between the server and the clients are handled using Node.js's socket.io library. The server is in charge of making requests to web services, storing data and sending the necessary data to the frontend editor. It also hosts web applications created in Gneiss, providing URLs for people to access the web applications without opening the frontend editor.

7.1.1 LIVE FRONTEND EDITOR IMPLEMENTED WITH CONSTRAINTS

Gneiss uses a one-way constraint library called ConstraintJS [70] to maintain the dependencies among various UI elements in the frontend editor. The ConstraintJS library lets developers create constraint objects that can be functions that use other constraint objects to return dynamic values, and bind a constraint's return value to a web GUI element such as the text of a text label. When a constraint changes its value, ConstraintJS automatically propagates the changes to other depending constraints and updates them in turn. In Gneiss, I use different types of constraints to implement the source pane, the spreadsheet and the web interface builder, and to trigger new data requests and handle the return data:

- Web API constraints: A web API constraint sends a web API to Gneiss's server and waits for the server to return values. A web API constraint is re-evaluated (resending the web API to the server) when its API is in the URL bar in the source pane and the user hits the refresh button, and when any spreadsheet cells this API uses change their value. The constraint returns a special value "Loading…" when it is still waiting for the server, and the

returned JSON document when the server returns. For streaming data, the web API constraint also sends the time range of the desired data (if the user sets rules to sort or filter the data by time) to the server to help narrow down the size of the return data.

- Source pane constraint: The source pane constraint uses the web API constraint whose API is currently in the URL bar. Its returned value is bound to the returned data area in the source pane (Figure 4.2 at 1). If the web API constraint returns "Loading…", the source pane constraint also returns "Loading…". If the web API constraint returns a JSON document, the source pane constraint turns the document into a string, adds HTML tags to the string for styling and returns the string.

- Hierarchical table constraints: Recall that in Gneiss, adjacent spreadsheet columns that use data from the same source (either a web service or a local file) form a hierarchical table that can be visualized and manipulated using the relative hierarchical relationships among the columns (see Chapter 4). To implement this, internally each hierarchical table is a JSON object that is created using the source document and information about the columns, such as a column's data path and sorting and filtering rules. A hierarchical table constraint creates this JSON object for a hierarchical table. The constraint is a function that takes a copy of the source document and the column information to extract the required fields, organizes them into a new structure and returns the final object. Every time that the source document changes (such as when the user makes a new web API query) or when the column information changes (such as when the user drags a column to a different location, or sets a new sorting rule for a column), the hierarchical table constraint re-evaluates and returns a new JSON object for this table. Details about the structure of the internal JSON objects for hierarchical tables are described in section 7.2 below.

- Spreadsheet cell data constraints: Each spreadsheet cell in Gneiss has a cell data constraint object. The constraint is a function that takes a string of spreadsheet code that is entered into this cell. The function uses a parser that I implemented to translate spreadsheet code into JavaScript code. If the spreadsheet code is a formula (a string starting with an equal sign), the parser replaces all spreadsheet cell names in the string with their corresponding cell data constraint names. The function then uses a native JavaScript function `eval(exp)` to execute the translated JavaScript code, and returns the results.

  If the spreadsheet cell data come from a data source, either a web service or a local file, its input spreadsheet code string is a special function

`=getHierarchicalTableData(hierarchicalTableID, cellName)`. This function uses the returned JSON object of a hierarchical table constraint and returns the data for the cell using `cellName`. The function is not exposed to the user as a cell whose value is dragged from the source pane is not editable (but can be cleared by right-clicking the cell and selecting "clear content").

- Spreadsheet cell visualization constraints: Every spreadsheet cell also has a cell visualization constraint that takes the return value of its cell data constraint and turns it into a HTML string for display. The return value of the cell visualization constraint is bound to the spreadsheet cell UI element in Gneiss's spreadsheet editor. If the input is a plain string that does not have any structure, the visualization constraint returns the plain string. If the input is a JSON object, the visualization constraint returns a nested table in HTML based on the structure of the object.

- Web GUI property constraints: As explained in chapter 4, Gneiss has a web interface builder that lets the user create web pages where elements in the web page can use spreadsheet formulas as their attribute values. Each attribute of a web interface element created in Gneiss (such as the color of a text label or the value of a textbox) has a web GUI property constraint object. A web GUI property constraint works similarly to a spreadsheet cell data constraint. It takes a string of spreadsheet code and returns the execution of the code. The return value of a web GUI element property constraint is bound to the corresponding property of a web GUI element in Gneiss's web interface builder.

- Streaming web UI constraints: This is a special type of constraint that streams data from a web element in the web interface builder to Gneiss's server. A streaming web UI constraint is a function that takes a property value of a web element and a pausing condition. When the pausing condition changes to be false, the constraint sends the value of the web element property to Gneiss's server. If there is no pausing condition, then the constraint is re-executed (sending the property value to the server) every time when the property value changes. The constraint returns the data returned from the server. When waiting for the server's response, the constraint returns "Loading…" same as the web API constraint does.

These constraint objects link different parts of the client Gneiss editor together and also handle the communications between the client editor and Gneiss's server.

7.1.2 INTERACTING WITH WEB SERVICES AT THE SERVER

As described earlier, the web API constraints in the client-side editor send API requests to Gneiss's server, and the server further sends them on to the web services. When a non-streaming API request returns, the server adds a timestamp to the returned data and sends the returned data back to the client spreadsheet. This timestamp is the source of cell metadata that represents the fetched time of a cell whose value comes from a web service. Currently, data returned from non-streaming API requests are not stored in the server's database, since I assume that for non-streaming requests, the data values are not time-sensitive and therefore when the user needs the data, she can just retrieve it again. Whereas for streaming data, since the data are often live values and the user may want to analyze historical data, they are stored in the database.

For streaming data, the server periodically sends the web API request based on the streaming frequency set by the user in the frontend editor, and stores the returned data with a timestamp and the ID of the client spreadsheet to the database. If the streaming data source is not a web service but a web element property value, the server directly stores that value in the database together with the web element's ID and property name, a timestamp and the spreadsheet's ID. The server then retrieves all data from this streaming source in the database that have the same spreadsheet ID as the client and have been fetched within the time range the user has specified in the frontend editor, and sends only those data to the client. This allows the client machine to work on a smaller set of data and thus to have better performance. Currently, the server only sorts and filters the data by their timestamps and leaves other sorting and filtering operations to the client. Future work could be to move all the sorting and filtering work to the server to further decrease the amount of data sent back to the client.

7.1.3 SAVING AND REOPENING A SPREADSHEET

As mentioned previously, Gneiss lets the user save a spreadsheet and reopen it. When the user saves a spreadsheet, the system outputs a JSON file about the spreadsheet for the user to download that records the spreadsheet's ID, all the data sources used (web services and local documents), all the web elements created in the web interface builder, all the input values in spreadsheet cells and web GUI element properties, and all the sorting and filtering rules. Currently, the spreadsheet cannot be stored on the server. That is a future work. To reopen a spreadsheet, the user loads that JSON file into the Gneiss editor. Gneiss uses the file to recreate the constraint objects, rebuild the web pages created in the web interface builder, and retrieve the required data from web services (for non-streaming data) and Gneiss's database (for streaming data).

7.1.4 EXPORTING A WEB APPLICATION

Exporting a web application in Gneiss works similarly to saving and reopening a spreadsheet. When the user chooses to export a web application in Gneiss, the system collects the same information as saving the spreadsheet and stores that JSON object in the server's database. When the web application is launched, the system retrieves the spreadsheet JSON object from the database to recreate the application. On the web server, the system will create a new folder for the created application (which provides a URL to the application). In my current implementation, in that folder, the system generates a HTML file that has all the pages created in the web application. Each page is a big DIV element that includes all the web elements created in the page and has an ID that is the page name. Only the "index" page DIV element is visible when the application is first opened. Going to different pages in the web application is implemented as showing the DIV element of the specific page and hiding the rest of them.

A web application can be opened and used at multiple places at the same time. Each copy of the web application has its own copy of the client spreadsheet. Therefore, in most situations the same application opened at different places will not affect each other. For example, in the usage scenario in Chapter 4, what one user searches and sees in the application on one device does not affect what another user searches and sees in the application on another device. The only case where a user's interaction in an application may change how other users see this application later is when the user adds or removes streaming data on the server's database (as mentioned previously, streaming data in Gneiss are stored in the database and retrieved to the client using the spreadsheet ID).

In my current implementation, an exported web application has the same ID as its spreadsheet. Therefore, multiple copies of a web application will share the same database account. Sometimes this design creates the desired behavior. For example, in the web exam application described in the second demonstrative example in Chapter 6, every time when a new student completes a test, the answers are sent to the server and stored in the same database. Later when another student opens the web application, he will see an updated class average score that includes the data of the previous student, as the previous student's data has been added to the database. Note that if an application only lets users retrieve different sets of data from the database to the frontend but does not let users change data in the database, its copies will not affect each other. For example, the market price monitor described in the usage scenario lets users choose to visualize the data of today or the past five days. When the user changes the selection, the server retrieves different sets of data from the database and sends to the client, but does not add or delete any data in the database. Therefore, the client-side copies do not affect each other in this situation.

Data from non-streaming APIs are returned directly to the client and not stored in the database. Therefore different copies of an application that does not stream data from external sources (such as the search application in the usage scenario in Chapter 4) will never affect each other.

7.2 INTERNAL JSON DATA MODELS

As described earlier in Chapter 5, in Gneiss, adjacent spreadsheet columns whose data come from the same hierarchical document forms a hierarchical table. For example, in Figure 7.1, columns B – E form a hierarchical table as the data in these columns are from the same document returned from the same web service. A hierarchical table can be referenced, sorted and filtered using its structure. To enable this, for each hierarchical table in the spreadsheet, there is a corresponding JSON object that is built using the relative hierarchical relationship of the data among the spreadsheet columns. Internally, there is a system function that maintains the "profiles" of all hierarchical tables in the spreadsheet. The profile of a hierarchical table includes the columns that form this table, and for each of the columns, the column path to its data in the original document (for example, for column B in Figure 7.1, its column path is `$[*]["name"]` as it stores all restaurant names), and any sorting, filtering and grouping rules the user has set on this column. This function will check every time when the user drags a column to a different location or sets a new sorting, filtering and grouping rule, to see if any part of the hierarchical table's profile needs to be updated or if a new hierarchical table needs to be created. If so, the function will trigger the reconstruction of that table's JSON object. As described earlier, a hierarchical table is maintained by a constraint object. The constraint is a function that constructs a JSON object using the following steps:

B (name)	C (avg_rating)	D (address1)	E (reviews.text_excerpt)	
The Dor-Stop Restaurant	4.5	1430 Potomac Ave	1.1	Went here thanks to numerous recommendations from other yelpers. We were glad we did. Definitely small, so a wait should be expected on busy weekend...
			1.2	I was excited to check this restaurant because of the reviews and my love for carbs. I also saw that it was on Triple D, though usually those pursuits end...
			1.3	I have had this place on my bucket list for some time now. Today my girlfriend and I decided to go for brunch. We had a very difficult time parking but...
Waffallonia	4.5	1707 Murray Ave	2.1	This place is awesome. It's a hole in the wall, but it's awesome. I was a bit confused about the menu. But in my defense, I was way too excited about this...
			2.2	This place is just plain magic. Obviously it's very tiny and not really a sit down place but whatever -- their waffles are out of this world good. I went...
			2.3	This tiny little shop is absolutely amaaazing! It's a waffle dessert shop that serves sugar waffles and choice of toppings. A great place to hang out and...

Figure 7.1. A hierarchical table visualized in nested cells in Gneiss.

- The function first creates an empty JSON object with one array field whose name is the name of the leftmost column. It then collects all values of this column from the original document using the column's path, and creates the same number of items to put in the array. Each array item has two fields: a `value` field that stores the value, and a `cell_path` field that stores the path to this value. The function also stores the name of this column in a temporary variable called `root_array`.

For example, in Figure 1, column B is the start of a hierarchical table. The system creates a JSON object like this:

```
{"B":[{"value": "The Dor-Stop Restaurant",
       "cell_path": "$[0]['name']"},
      {"value"; "Waffolonia",
       "cell_path": "$[1]['name']"},
    …
 ]}
```

`root_array` is now "B".

- Then for the rest of the columns, the system compares a column's path to its immediate left column's path.

    - If this column's path is at the same array level as the immediate left column's path, such as in Figure 1 where column C's column path (`$[*]["rating"]`) is in the same array level as column B's column path (`$[*]["name"]`), the system creates a new object whose name is the name of this column inside each item in `root_array`. The new object also has a `value` and a `cell_path` field. The system uses the `cell_path` field in the `root_array` to construct the `cell_path` for the new object and to get the new value, making sure that data in the same item in `root_array` come from the tree branch in the source document. To do so, the system first finds the common path of the two paths and then constructs a new path by replacing the common path in the new column with the array index of the element in the root array.

      For example, in Figure 1, the common path of column C's column path (`$[*]['rating']`) and the `root_array` (column B)'s column path (`$[*]['name']`) is `$[*]`. When creating a new object in the first item in the root array B, the system replaces `$[*]` in its column path with `$[0]` that is the array index of the common path in the `cell_path` in the root array object (`$[0]['name']`) to form the `cell_path` for the object C (`$[0]['rating']`) and retrieve the value 4.5 from the source document. The JSON object after processing column C becomes:

      ```
      {"B":[{"value": "The Dor-Stop…"
             "cell_path": "$[0]['name']"
             "C":{"value": 4.5,
                  "cell_path": "$[0]['rating']"
      ```

```
            }
        },
        …
 ]}
```

`cell_path` in the first item in `root_array` (B) starts at array index 0. So the `cell_path` for the new object C in the first item also starts at array index 0.

- If this column's path is at a deeper array level than the immediate left column's path, the system creates a new array whose name is the name of this column inside each item in `root_array`. Each item in the new array also has a `value` and a `cell_path` field. Again, the system uses the `cell_path` field in items in the `root_array` item to construct the `cell_path` for each item in the new array. After the system updates the JSON object, `root_array` become the current column.

  Continuing the previous example, in Figure 1, column E's column path (`$[*]['reviews'][*]['text_expert']`) is in a deeper array level than column D's column path (`$[*]['address']`). The common path of the two paths is again `$[*]`, and the `cell_path` in the first item in `root_array` (B) starts at array index 0. Therefore, the array D in the first item in B has the path `$[0]['reviews'][*]['text_expert']`. The JSON object after processing column E becomes:

```
{"B":[{"value": "The Dor-Stop…"
       "cell_path": "$[0]['name']"
       "C":{"value": 4.5,
            "cell_path": "$[0]['rating']"
           },
       "D":{"value": "1430 Potomac Ave",
           "cell_path": "$[0]['address']"
           },
       "E":[{"value":"Went here thanks…",
             "cell_path":"$[0]['reviews']
                         [0]['text_expert']"},
            {"value":"I was excited to…",
             "cell_path":"$[0]['reviews']
                         [1]['text_expert']"},
             …
           ]
       },
     …
```

```
]}
```

`root_array` is now "D".

- Finally, if this column's path is at an upper array level than its immediate left column's path, the system creates a new object whose name is the name of this column inside each item in `root_array`. Similarly, each new item has a `value` and a `cell_path` field. Again, the system uses the `cell_path` field in items in the `root_array` to construct the `cell_path` for each item in the new object and get the new value.

  Continuing the previous example, in Figure 1, suppose the user drags the country field of the restaurants to column F. Column F's path now becomes `$[*]['country']`, which is in the upper level array compared to column E (`$[*]['reviews'][*]['text_expert']`). So the system creates an object name "F" in each item in array E. For the first item in array E in the first item in array B, its `cell_path` starts at index 0 (`$[0]['reviews'][0]['text_expert']`). Therefore, the `cell_path` in the new object F in the first item in array E in the first item in array B becomes `$[0]['country']`. Now when we look at the second item in array E in the first item in array B, its `cell_path` also starts at index 0 (`$[0]['reviews'][1]['text_expert']`). So the `cell_path` new object F in the second item in array E in the first item in array B is still `$[0]['country']`, creating a duplicate value. The JSON object after processing column F becomes:

```
{"B":[{"value": "The Dor-Stop…"
       "cell_path": "$[0]['name']"
       "C":{"value": 4.5,
            "cell_path": "$[0]['rating']"
           },
       "D":{"value": "1430 Potomac Ave",
           "cell_path": "$[0]['address']"
           },
       "E":[{"value":"Went here thanks…",
             "cell_path":"$[0]['reviews']
                          [0]['text_expert']",
             "F":{"value":"USA"
                  "cell_path":"$[0]['country']"}
             },
             …
           ]
```

```
            },
          …
    ]}
```

Once the system finishes building this JSON object, sorting, filtering and grouping using the object become straightforward. Sorting by a column affects only data that is at or below the current array level. For example, sorting by column E (review text) will change the item order in every array E using the `value` field in each item. As object F is in an item in array E, it is being moved as the items are moved in array E, causing data in column F in the spreadsheet to also change. But this does not affect the item order in array B, which is the parent array of array E. Sorting by column C (ratings), however, will change the item order of array B, as object C is directly in array B.

Filtering by a column will cause the array item that the column object is in to be deleted if the `value` field does not match the filtering criteria. For example, if the user filters the data by column C (ratings) and sets the criteria to be <4, the first array item in array B will be removed as the `value` field of object C is 4.5. If a column is selected for grouping, its next (right) column will be become an array (if not one already) to store the merged values. Every time an internal JSON object changes, all spreadsheet cells linked to this object will refresh and show the latest data.

7.3 CONCLUSIONS

Gneiss is implemented as a web application and uses a client-server architecture. The client editor uses various constraint objects to provide a live programming environment and connect the created web application to the spreadsheet editor and the external data sources. To visualize hierarchical data in the spreadsheet and allow the data to be manipulated by its structure, the client editor dynamically constructs JSON objects from the source document that represent the hierarchical relationships among data in spreadsheet columns. Gneiss's server is in charge of communicating with web services, storing streaming data and hosting the exported web applications. Gneiss is open-source and available at http://www.cs.cmu.edu/~shihpinc/gneiss.html.

CHAPTER 8 FUTURE WORK

I have presented how Gneiss extends conventional spreadsheets to enable using more types of online data and creating interactive, data-driven web applications. In this chapter, I discuss future work.

8.1 GNEISS EDITOR FEATURES

Gneiss is a research prototype that focuses on experimenting with new research ideas. As described in section 3.2, I implemented my own spreadsheet and web interface builder to give myself the most freedom to try out what I wanted. But as a result, the current Gneiss editor lacks many features that are common in conventional spreadsheets or web editors. I have mentioned much future work in previous chapters that could make Gneiss closer to a product-level type of tool. Here, I summarize those features again.

8.1.1 GENERAL USABILITY FEATURES

Gneiss lacks some common usability features, such as redo/undo, formatting cells in the spreadsheet, copying and pasting multiple cells in the spreadsheet, options to paste data by values or with formatting, highlighting cells when entering their names in a spreadsheet formula, copying and pasting GUI elements in the web interface builder, resizing or changing the locations of the three panes in Gneiss, etc. Therefore, obvious future work includes improving the usability of Gneiss by adding those features and letting people access them in the same was as in conventional editors, such as using a "ribbon" design like in Excel to display all the widgets/buttons for text formatting. Also, as mentioned in section 3.2, Gneiss's spreadsheet currently shows the top 50 rows of data and lets users view the entire data in a read-only spreadsheet. Future work can be implementing a "Next Page" button like Google Fusion Tables to let users view the next set of data to enable a more fluid browsing experience.

8.1.2 SPREADSHEET FUNCTIONS

Gneiss currently supports a limited set of spreadsheet functions (see Appendix A). Adding more spreadsheet functions to Gneiss is future work. Gneiss introduces several new spreadsheet functions (such as `TIMER` and `ANIMATION`) as well as extends some conventional functions (such as having `NOW` automatically update every minute). The usability of those new or extended functions could still be improved, such as allowing the user to customize the updating frequency of the `NOW` function through a parameter. Also, as mentioned in section 3.4.5, while Gneiss introduces nested cell representation, when multiple cells are selected to use in a spreadsheet function, their structure is flattened and the cells are put into a flat array to allow

them to be used in many conventional functions. Future work can be to design spreadsheet functions that can make use of the nested structures in data to provide new computational abilities.

8.1.3 SPREADSHEET SORTING AND FILTERING

Sorting and filtering data by a column in Gneiss is done using a dialog box opened by clicking on an arrow icon at the top of the column. I designed this based on Google Spreadsheet's design that when hovering on the top label of a column, an arrow icon appears and clicking on the icon will bring up a menu that has options for sorting and other operations. Other design can be used to make the sorting and filtering options more visible to the users. For example, Excel uses a toolbar at the top of the window to show icons for sorting and filtering. Gneiss could use this design. Currently in Gneiss, data retrieved from the same source are sorted and filtered together. Future work can be to let users customize the range for sorting and filtering. There are known ways in conventional spreadsheets to do that, which Gneiss can adopt.

In addition, currently the options for streaming (setting streaming frequency and pausing conditions) are put together with the sorting and filtering options in the dialog box (Figure 6.1 at 2). If Gneiss uses a UI similar to Excel that has a top tool bar and ribbons, the streaming options could become icons in the toolbar like sorting and filtering to be more visible to the users.

8.1.4 SPREADSHEET AUTOFILLING

As mentioned in section 3.2, Gneiss's autofilling is not as intelligent as Excel's when inferring new values based on example values. Also, Gneiss introduces nested row labels for referencing nested cells. Autofilling nested rows could introduce new challenges. For example, suppose spreadsheet column A is a flat column, B is a first level nested column, C is a second level and D is a third level. If the user enters `=A1+B1.1+C1.1.1` in `D1.1.1.1`, then selects `D1.1.1.1` and drags down to `D1.1.1.2`, what should happen? Currently, Gneiss uses a simple heuristic: it changes the row index that is in the same level as the autofilled cell. In this example, since the autofilled cell has three nested levels and all cells used in the formula have less than three nested levels, Gneiss will fill in the same formula `=A1+B1.1+C1.1.1` for `D1.1.1.2`. But maybe the user wants something different and needs to provide more examples. The `$` operator in conventional spreadsheets to have a column or row fixed when autofilling currently does not work in Gneiss. Improving Gneiss's current autofilling algorithm (such as adopting more of Excel's algorithm) and further extending it to better support inferring nested row indexes could be interesting future work.

8.1.5 WEB INTERFACE BUILDER FEATURES

As mentioned in section 3.2, Gneiss's web interface builder supports a limited number of GUI elements and properties (Appendix B). Adding more kinds of GUI elements, visualizations and properties is future work. Currently, GUI element IDs in Gneiss are assigned by the system and are not editable by the users. Allowing users to customize the ID of an element is also future work. Many conventional web interface builders (such as Adobe Dreamweaver) provides a "code view" that let users switch from editing the rendered web page to directly editing its source code to give skilled users more control over the look of the page. Gneiss's web interface builder could be extended to support a "code view". If so, how the spreadsheet statements written by the users should be displayed inline with HTML and CSS code of a web page can be an interesting future work. Benson et al.'s Quilt [10] puts the user's spreadsheet statements in HTML attributes, but there could also be other designs. Another approach is to let people import HTML and CSS files written elsewhere into Gneiss's web interface builder to program data bindings and interactive behaviors in spreadsheet languages.

8.1.6 ACCEPTING MORE TYPES OF DATA SOURCES

As mentioned multiple times in previous chapters, Gneiss's source pane currently supports REST web services that return JSON data and local JSON files. Future work can be extending the source pane to support more data types. For example, to let users import CSV files or Excel spreadsheets to use in Gneiss's spreadsheet, to extend the definition of hierarchies to support hierarchical XML files, or to further support extracting data from HTML pages as in prior work such as [28,87]. As I described in section 3.6, I expect most of the novel techniques presented in this dissertation can still work for these new data sources with little modification.

8.2 SERVER-SIDE EXTENSIONS

As described in the previous chapter, Gneiss uses a client-server architecture. The client is the editor interface, and the server handles the communication with web services and does data storage.

8.2.1 MOVING CLIENT-SIDE COMPUTATIONS TO THE SERVER

Currently, all the hierarchical data reshaping, restructuring, sorting, filtering and grouping described in section 7.2 happen in the client-side Gneiss editor. Depending on the size of the data and the resources of the client machine, this manipulation process could take a couple of seconds, causing a noticeable delay in the interface. A possible way to improve Gneiss's performance is to move this data manipulation process to the server. While this approach decreases the computation load of the client, it will increase the network transmission load and add the network

transmission time, as the data will be passed more often between the server and the client. One solution is to let the user be able to choose whether to run the data manipulation on her own machine or to upload the data to Gneiss's server and run there. This could depend on the size of the data, the user's machine and the network condition. If the data manipulation is set to run on the server, many known techniques can be used on the server to speed up the computation, such as using map-reduce and parallel computing to improve performance.

8.2.2 SHARING VS. INDEPENDENT WEB APPLICATION DATA

As described in section 7.1.4, currently different copies of a web application that uses streaming data may affect each other if one copy adds or removes data in the database on Gneiss's server. Different copies of a web application that do not use streaming data are independent. Currently, this behavior is not explained in the interface to the user, and the user has no control over that. Exposing this behavior to users and allowing them to choose how the data are shared among applications is future work.

8.3 IMPROVING THE PROGRAMMING OF INTERACTIVE BEHAVIORS

As discussed in section 4.5, the way that Gneiss currently supports programming interactive behaviors in web applications using once-around constraints and interactive properties has several limitations. For example, the user cannot programmatically set the input value of a GUI control (such as a value of a textbox or a checkbox), as the input value is bound to an interactive property whose value only changes based on how people interact with it on the web page. The use of once-around circular constraints enables programming more complicated behaviors such as detecting the last-clicked button (such as in the usage scenario in section 4.2) or incrementally increasing a cell's value using self-referencing (such as in the demonstrative example in section 4.4.1). However, they are a little tricky to program. Since once-around constraints are not common in conventional spreadsheet systems (for example, Excel by default does not support once-around constraints), it is doubtful that spreadsheet users (especially those who do not have much web programming experience) would know how to use (or even ever think of using) once-around constraints to program these behaviors. Prior spreadsheet systems such as Forms/3 [11] and InterState [71] have used different approaches to support programming interactive behaviors in graphical user interfaces, but their approaches are not designed for non-programmers.

Therefore, a direction for future work can be to improve Gneiss to support programming different types of web interactive behaviors in a way that is more intuitive to spreadsheet users. An immediate improvement I can think of is to provide higher-level spreadsheet functions for programming some common

interactive behaviors. For example, a `LASTCLICKED(GUIIDs)` function might take a list of GUI element IDs and return the ID that was last clicked. An advantage of spreadsheet programming for end-users is the higher level functions that support common data-manipulation behaviors to avoid many lower level programming details, such as using a `SUM` function to sum a list of values instead of writing a for-loop and adding the values one by one [69]. So future work can be studying the most common interactive behaviors in data-driven applications and designing higher-level spreadsheet functions with naming and input parameters easily understandable to spreadsheet users to support programming those behaviors. The once-around constraint evaluation method can co-exist with the higher-level interactive behavior functions to support programming custom behaviors that more advanced users might want.

8.4 SPREADSHEETS AS DATABASE CONSOLES

As described in Chapter 6, Gneiss contributes a way to select the desired data from a document, group data, join data and calculate summaries of data in a spreadsheet using only interaction techniques and spreadsheet functions without requiring the user to learn other programming concepts such as pivot tables or SQL. These behaviors together with the familiar sorting and filtering ability of spreadsheets are also popular database commands. In Gneiss, many database commands are replaced by interaction techniques in the spreadsheet. For example, the `SELECT…FROM…WHERE…ORDER BY…` database statements can be done by drag-and-dropping fields to the spreadsheet and refining through sorting and filtering (as in the scenario in section 3.4); and the `SELECT…FROM…WHERE…GROUP BY…` statements can be done by rearranging the columns, grouping them and using spreadsheet functions to compute aggregate values (as in the scenario in section 6.2). Therefore, a direction for future work can be using Gneiss as a database console to generate queries to a database using spreadsheet mechanisms. This is essentially creating a map between various interaction techniques supported in Gneiss and the database commands. While there are conventional query builders that let users build SQL queries using GUI widgets and block diagrams (such as Microsoft Access's Query Design [103] and Active Query Builder [117]), Gneiss provides an opportunity to construct those queries in a familiar spreadsheet environment using only spreadsheet mechanisms. Unlike in traditional query builders where the user constructs a long query using widgets and hits run to see the data, if using Gneiss, the user would build a query in much smaller steps and for each step the user could see the resulted data thus providing incremental visual feedback, which could be useful for novice and even expert users who find it difficult to write a long query at once. Prior research on incrementally querying of a database to provide interactive visual feedback (e.g., [33]) could be a way to allow the system to have reasonable performance when querying over a large amount of data. Another research direction is to provide *all* SQL operations using the Gneiss-style mechanisms.

8.5 DATA-DRIVEN MOBILE APPLICATIONS

Gneiss introduces a way to use spreadsheet languages to program data bindings between web GUI elements and local or online data sources, and interactive behaviors that let people query, sort and filter to see the information they want. An interesting direction is to extend Gneiss's programming paradigm to support creating native mobile applications that let people use and interact with online data. Prior research has shown that being able to receive information while on the go is one of the main reasons why people use mobile devices, and people have all kinds of information needs but many of them are not addressed by existing applications [79]. Using familiar spreadsheet programming could be one way to facilitate the creation of mobile information applications. While Gneiss currently supports creating web applications that can be opened on any devices that have a browser, mobile applications are in general more powerful and usable than web applications when running on mobile devices. Programming a mobile application introduces new programming challenges. For example, conventional mobile devices provide many sensors that are often used in mobile applications to provide feedback related to the context of the environment, such as sorting a list of places by their distances. Mobile applications can also make use of operating system features such as popping up notifications on the home page.

I can see some of Gneiss's features being extended to support programming these new activities for mobile applications. For example, the way Gneiss supports streaming data from web services could be used to support streaming sensor readings on mobile devices, since sensor values are also real-time. The spreadsheet's live programming could be useful for generating notifications that tell the user that something just changed. Spreadsheets have a known way to create data visualizations and can be useful to create dashboards that are popular on mobile devices for viewing data. Some problems are more difficult to address. For example, a big advantage of a live programming environment is to allow programmers to quickly switch between programming and testing. Unlike web applications, mobile applications need to be compiled and deployed onto a mobile device for testing. Further, Gneiss's design makes use of multiple panels that might be hard to use on a small smartphone screen. How to extend Gneiss's live programming and programming-with-example style to support programming mobile applications on the mobile devices could be an interesting future research topic.

8.6 DEBUGGING

Gneiss extends spreadsheets into new domains. A spreadsheet can now hold hierarchical and streaming data from web services. A spreadsheet can also contain

GUI element values from a web application. With these new extensions, what are some possible programming errors that the user might make, and how can the system help the user debug remain open questions. Spreadsheets are known to be difficult to debug because the user by default sees the value of all cells instead of the code that generates the values. Conventional spreadsheets such as Excel provide some debugging help such as letting users view the formulas of the cells instead of the computed values, highlighting cells used in a formula when editing the formula, detecting unusual patterns in the spreadsheet and warning users, for example, if there is only one cell in a row that is not computed using a formula. Gneiss could benefit from having those features, but more novel approaches could be investigated. For example, having a good "design pattern" to separate the web service data and GUI element properties into multiple spreadsheets could be one way to make the spreadsheets more maintainable.

8.7 INCORPORATING GNEISS INTO CURRENT PROGRAMMING PRACTICES

Gneiss introduces a new way to program data-driven applications using spreadsheets in a visual programming environment. This is different from the conventional way that programmers write textual code in a text editor. I experimented with many new ideas in Gneiss to support different programming tasks. It is very likely that a person may find that some of Gneiss's new features are very useful for her, while others are not really relevant, and some of her tasks are not supported by Gneiss at all. Therefore, I think one way to generalize the results of this work would be to look at how different parts of the innovations introduced in Gneiss could be incorporated into people's current programming practices.

For example, Gneiss allows people to use the spreadsheet as a database for a web application and program two-way data flows among web GUI elements, spreadsheets and web services using the familiar spreadsheet languages. Research has found that while many people know how to program static web pages using HTML and CSS, programming web pages that use backend data and present dynamic content requires writing other languages such as JavaScript and is much more difficult [75]. Therefore, a possible way to bring Gneiss's technology to more people could be to allow people to import HTML and CSS files into Gneiss's right pane to program data bindings, communications with web services, and data-related interactive behaviors using Gneiss, and leave the styling part to writing HTML and CSS code that may give users more control than using an interface builder.

Another topic comes from the way that Gneiss provides a visual environment to work with JSON data. It introduces a new way to visualize JSON data in spreadsheets that lets users regroup and reshape the data easily using interaction techniques and calculates summaries of data using spreadsheet formulas. Our user study showed that Gneiss helped spreadsheet users outperform programmers writing code in

doing data exploration tasks that involve restructuring a JSON object and joining multiple objects. However, in real life, many tasks that involve using JSON data are beyond the capability of a spreadsheet. For example, the programmer participants of my user study used JSON data to run statistical analysis and create mathematical models. All of them mentioned that there were a lot of data restructuring when using JSON data, sometimes by feeding the data into another function, sometime designed to help them understand the data (such as calculating summaries like mean values). A way to incorporate Gneiss's technology into programmers' current workflow could involve letting the programmer manipulate a JSON object in Gneiss and output the corresponding code to do these operations in another language such as JavaScript or Python, so the programmer can use the code in his other programs. Another interesting direction is to use Gneiss's mechanisms to create a new web service that takes input parameters as specified by the user in spreadsheet cells and returns data in spreadsheet columns as a JSON object.

8.8 UNDERSTANDING END-USER PROGRAMMERS IN THE REAL WORLD THROUGH GNEISS

I have conducted an initial user study that evaluated Gneiss in a lab setting. The user study showed that users could understand and use the nested table visualizations for hierarchical data, the new spreadsheet language syntax for referencing data using its structure, and the interaction techniques for reshaping, regrouping and joining data. The study also showed that those features together helped users manipulate hierarchical data more efficiently compared with conventional methods. The user study also tested the usability of some other features in Gneiss including the design of the source pane, the drag-and-drop gesture for extracting data to spreadsheets, and using the dialog box for sorting and filtering, showing that these features were in general learnable and usable.

But there are some other features in Gneiss that have not been evaluated yet by any lab study. Some untested features are extensions of current mechanisms and therefore it may be safe to assume that users can learn how to use them. For example, Gneiss's right pane is similar to a conventional web interface builder, and it uses conventional spreadsheet language syntax for selecting a cell in a spreadsheet (`SpreadsheetName!CellName`) to support selecting a property of a GUI element (`GUIElementID!PropertyName`). Another example is that Gneiss provides a cell's fetched time as its metadata that is hidden and thus does not add additional complexity to the editor's interface, but allows the metadata to be used in conventional ways such as in functions, sorting and filtering to enable manipulating the data by time. However, other features may require further testing. For example, as discussed in the section 8.1, Gneiss allows once-around circular constraints in spreadsheets and introduces new `TIMER` and `ANIMATION` functions to support programming interactive behaviors in web applications. To what extend those

features can be utilized by spreadsheet users who have never programmed a web application before still needs to be evaluated. Running a few more lab studies to fully examine the new features of this dissertation is a future work that can help researchers further understand the strengths and limitations in Gneiss's design and improve the usability of the features.

A more interesting type of study is to deploy Gneiss in the real world, and see what real users create with Gneiss and how. As discussed in Chapter 2, Gneiss is motivated by prior literature which showed that end-users have the need to create custom database or data-driven web applications [9,75]. While many prior systems allowed end-users to publish their own datasets on the web, none of them supported creating completely custom interfaces and interactive behaviors for showing and using data as Gneiss does. Gneiss uses the familiar spreadsheet model, which could potentially attract more users compared with prior systems that use other programming models. Deploying Gneiss in the wild and studying its users and the artifacts created can provide more insights about end-user programmers and their needs for data-related tools. Currently, I have published Gneiss as an open-source tool where people interested in the technology of Gneiss can download its source code, run it and make modifications on their own. But there is still much work required to harden Gneiss to make it a product-level tool that end-users could use for their everyday tasks, and there can be new contributions in the system architecture to make Gneiss available to a large amount of users. Real-world studies usually take much longer time to collect results than lab studies (for example, the Exhibit system was first published at 2007 [43] and its first real-world study after the deployment was published in 2014 [9]), but such studies also contribute valuable knowledge on end-users and their needs that lab studies cannot provide.

8.9 CONCLUSIONS

Gneiss raises some unanswered questions and opens up some interesting future research topics. Those topics range from further extending Gneiss to support more types of programming activities, to figuring out a way to connect Gneiss's technology with current programming practices, to deploying Gneiss in the real world as a platform to study end-user programmers.

CHAPTER 9 CONCLUSIONS

This dissertation contributes four main extensions to the spreadsheet model: (1) supporting constructing two-way data flow with REST web services; (2) programming interactive web applications that can dynamically use and modify spreadsheet data; (3) using structural hierarchical data such as JSON; and (4) using live streaming data. Multiple innovations in spreadsheet interfaces, spreadsheet languages and interaction techniques are introduced in this new model to support these activities.

This dissertation also contributes a tool called Gneiss that implements this new spreadsheet model. Throughout the dissertation, I have used Gneiss to provide a series of examples to demonstrate that with this new model, people can use the familiar spreadsheet languages and interaction techniques to program a variety of data-driven or data analysis applications that otherwise would require writing lots of code. I also presented evidence collected from a lab study that showed spreadsheet users who were not experienced programmers could use Gneiss to analyze hierarchical data significantly faster than spreadsheet users using Excel and programmers writing JavaScript or Python.

As I discussed the introduction, the Internet has made all kinds of public and personal data available to people. Those data have already been used by many professional programmers to gain insights or to create new applications to facilitate people's lives. I hope that by leveraging the familiar spreadsheet programming, this research could also empower end users to use online data in creative ways as they do right now with local data using spreadsheets. The success of spreadsheets has shown not only that end users have the need to make custom use of data, but also that end users can and will learn programming to do things themselves if they have the right tools. Gneiss shows that spreadsheets can be extended in interesting ways without needing to abandon the basic spreadsheet model and the interface that people are already familiar with and without needing to learn other concepts such as pivot tables, Visual Basic, JavaScript or SQL. Future research could build on Gneiss's contributions to let end user programmers be able to leverage what they know, rather than a hitting a "wall" [66] and having to learn entirely new things.

My ultimate hope is that the technologies presented in this dissertation could one day go beyond Gneiss to become a part of a real product that people use in real life, and bring us one step closer to the goal of empowering end users to freely use data, analyze data and publish their creative solutions just as professional programmers can.

REFERENCES

1. Robin Abraham and Martin Erwig. 2006. Inferring Templates from Spreadsheets. *Proceedings of the 28th International Conference on Software Engineering*, ACM, 182–191. http://doi.org/10.1145/1134285.1134312
2. Sam S Adams. 1988. MetaMethods: The MVC Paradigm. *HOOPLA, July*.
3. Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. 2011. The Extensibility Framework in Microsoft StreamInsight. *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, IEEE Computer Society, 1242–1253. http://doi.org/10.1109/ICDE.2011.5767878
4. Bryce Allen, John Bresnahan, Lisa Childers, et al. 2012. Software As a Service for Data Scientists. *Commun. ACM* 55, 2: 81–88. http://doi.org/10.1145/2076450.2076468
5. Pierpaolo Baglietto, Fabrizio Cosso, Martino Fornasa, et al. 2010. Always-on Distributed Spreadsheet Mashups. *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, ACM, 8:1–8:8. http://doi.org/10.1145/1944999.1945007
6. John Baker and Stephen J Sugden. 2007. Spreadsheets in education–The first 25 years. *Spreadsheets in Education (eJSiE)* 1, 1: 2.
7. Eirik Bakke, David Karger, and Rob Miller. 2011. A Spreadsheet-based User Interface for Managing Plural Relationships in Structured Data. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2541–2550. http://doi.org/10.1145/1978942.1979313
8. Eirik Bakke, David R Karger, and Robert C Miller. 2013. Automatic Layout of Structured Hierarchical Reports. *IEEE Transactions on Visualization and Computer Graphics* 19, 12: 2586–2595. http://doi.org/10.1109/TVCG.2013.137
9. Edward Benson and David R Karger. 2014. End-users Publishing Structured Information on the Web: An Observational Study of What, Why, and How. *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, ACM, 1265–1274. http://doi.org/10.1145/2556288.2557036
10. Edward Benson, Amy Zhang, and David R Karger. 2014. Spreadsheet-Driven Web Applications. *ACM symposium on User interface software and technology*, ACM, To appear.
11. Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. 2001. Forms/3: A First-order Visual

Language to Explore the Boundaries of the Spreadsheet Paradigm. *J. Funct. Program.* 11, 2: 155–206. Retrieved from http://dl.acm.org/citation.cfm?id=968486.968487

12. Margaret Burnett, Sudheer Kumar Chekka, and Rajeev Pandey. 2001. FAR: an end-user language to support cottage e-services. *Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on*, 195–202. http://doi.org/10.1109/HCC.2001.995259

13. Margaret M Burnett, John W Atwood Jr., and Zachary T Welch. 1998. Implementing level 4 liveness in declarative visual programming languages. *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, 126–133. http://doi.org/10.1109/VL.1998.706155

14. Chris Chambers and Chris Scaffidi. 2010. Struggling to Excel: A Field Study of Challenges Faced by Spreadsheet Users. *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, 187–194. http://doi.org/10.1109/VLHCC.2010.33

15. Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, and Pat Hanrahan. 2008. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. *IEEE Transactions on Visualization and Computer Graphics* 14, 6: 1213–1220. http://doi.org/10.1109/TVCG.2008.178

16. Kerry Shih-Ping Chang, Brad A Myers, Gene M Cahill, Soumya Simanta, Edwin Morris, and Grace Lewis. 2013. Improving Structured Data Entry on Mobile Devices. *ACM symposium on User interface software and technology*, to appear.

17. Kerry Shih-Ping Chang, Brad A Myers, Gene Cahill, Soumya Simanta, Edwin Morris, and Grace Lewis. 2013. A Plug-in Architecture for Connecting to New Data Sources on Mobile Devices. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 51–58.

18. Kerry Shih-Ping Chang and Brad A Myers. 2012. WebCrystal: Understanding and Reusing Examples in Web Authoring. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 3205–3214. http://doi.org/10.1145/2207676.2208740

19. Kerry Shih-Ping Chang and Brad A Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ACM, 87–96. http://doi.org/10.1145/2642918.2647371

20. Kerry Shih-Ping Chang and Brad A Myers. 2014. A spreadsheet model for using web service data. *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 169–176. http://doi.org/10.1109/VLHCC.2014.6883042

21. Kerry Shih-Ping Chang and Brad A Myers. 2015. A Spreadsheet Model for

Handling Streaming Data. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ACM, 3399–3402. http://doi.org/10.1145/2702123.2702587

22. Kerry Shih-Ping Chang and Brad A Myers. 2016. Using and Exploring Hierarchical Data in Spreadsheets. *ACM CHI*.

23. Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. 2013. Senbazuru: A Prototype Spreadsheet Database Management System. *Proc. VLDB Endow.* 6, 12: 1202–1205. http://doi.org/10.14778/2536274.2536276

24. Zhe Chen and Michael Cafarella. 2013. Automatic Web Spreadsheet Data Extraction. *Proceedings of the 3rd International Workshop on Semantic Search Over the Web*, ACM, 1:1–1:8. http://doi.org/10.1145/2509908.2509909

25. Zhe Chen and Michael Cafarella. 2014. Integrating Spreadsheet Data via Accurate and Low-effort Extraction. *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 1126–1135. http://doi.org/10.1145/2623330.2623617

26. Petr Chmelar, Radim Hernych, and Daniel Kubicek. 2008. Interactive visualization of data-oriented XML documents. In *Advances in Computer and Information Sciences and Engineering*. Springer, 390–393.

27. Eun Kyoung Choe, Nicole B Lee, Bongshin Lee, Wanda Pratt, and Julie A Kientz. 2014. Understanding Quantified-selfers' Practices in Collecting and Exploring Personal Data. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 1143–1152. http://doi.org/10.1145/2556288.2557372

28. Mira Dontcheva, Steven M Drucker, David Salesin, and Michael F Cohen. 2007. Relations, Cards, and Search Templates: User-guided Web Data Integration and Layout. *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, ACM, 61–70. http://doi.org/10.1145/1294211.1294224

29. Mira Dontcheva, Steven M Drucker, Geraldine Wade, David Salesin, and Michael F Cohen. 2006. Summarizing Personal Web Browsing Sessions. *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, ACM, 115–124. http://doi.org/10.1145/1166253.1166273

30. Robert J Ennals and Minos N Garofalakis. 2007. MashMaker: Mashups for the Masses. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ACM, 1116–1118. http://doi.org/10.1145/1247480.1247626

31. Gordon Filby. 2013. *Spreadsheets in science and engineering*. Springer Science & Business Media.

32. Danyel Fisher, Badrish Chandramouli, Robert DeLine, et al. 2014. *Tempe: An Interactive Data Science Environment for Exploration of Temporal and Streaming Data*. Microsoft Research. Retrieved from http://research.microsoft.com/apps/pubs/default.aspx?id=232385

33. Danyel Fisher, Igor Popov, Steven Drucker, and M.c. Schraefel. 2012. Trust Me, I'M Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 1673–1682. http://doi.org/10.1145/2207676.2208294

34. Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. 2004. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, ACM, 175–184. http://doi.org/10.1145/1029632.1029664

35. Michael I Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGARCH Comput. Archit. News* 34, 5: 151–162. http://doi.org/10.1145/1168919.1168877

36. Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 317–330. http://doi.org/10.1145/1926385.1926423

37. William R Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 317–328. http://doi.org/10.1145/1993498.1993536

38. Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R Klemmer. 2007. Programming by a sample: rapidly creating web applications with d.mix. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM, 241–250. http://doi.org/10.1145/1294211.1294254

39. Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2010. Automatically Extracting Class Diagrams from Spreadsheets. *Proceedings of the 24th European Conference on Object-oriented Programming*, Springer-Verlag, 52–75. Retrieved from http://dl.acm.org/citation.cfm?id=1883978.1883984

40. Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2011. Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 451–460. http://doi.org/10.1145/1985793.1985855

41. Scott E Hudson. 1994. User Interface Specification Using an Enhanced

Spreadsheet Model. *ACM Trans. Graph.* 13, 3: 209–239. http://doi.org/10.1145/195784.195787

42. Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. *Hum.-Comput. Interact.* 1, 4: 311–338. http://doi.org/10.1207/s15327051hci0104_2

43. David F Huynh, David R Karger, and Robert C Miller. 2007. Exhibit: Lightweight Structured Data Publishing. *Proceedings of the 16th International Conference on World Wide Web*, ACM, 737–746. http://doi.org/10.1145/1242572.1242672

44. David F Huynh, Robert C Miller, and David R Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, ACM, 125–134. http://doi.org/10.1145/1166253.1166274

45. Brian Johnson and Ben Shneiderman. 1991. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. *Visualization, 1991. Visualization '91, Proceedings., IEEE Conference on*, 284–291. http://doi.org/10.1109/VISUAL.1991.175815

46. Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-centred Approach to Functions in Excel. *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ACM, 165–176. http://doi.org/10.1145/944705.944721

47. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 3363–3372. http://doi.org/10.1145/1978942.1979444

48. Eser Kandogan, Eben Haber, Rob Barrett, Allen Cypher, Paul Maglio, and Haixia Zhao. 2005. A1: End-user Programming for Web-based System Administration. *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, ACM, 211–220. http://doi.org/10.1145/1095034.1095070

49. Max Van Kleek, Daniel A Smith, Heather S Packer, Jim Skinner, and Nigel R Shadbolt. 2013. Carpé Data: Supporting Serendipitous Data Integration in Personal Information Management. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2339–2348. http://doi.org/10.1145/2470654.2481324

50. Andrew J Ko, Robin Abraham, Laura Beckwith, et al. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3: 21:1–21:44. http://doi.org/10.1145/1922649.1922658

51. Woralak Kongdenfha, Boualem Benatallah, Julien Vayssière, Régis Saint-Paul,

and Fabio Casati. 2009. Rapid Development of Spreadsheet-based Web Mashups. *Proceedings of the 18th International Conference on World Wide Web*, ACM, 851–860. http://doi.org/10.1145/1526709.1526824

52. Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 5–8. http://doi.org/10.1109/VLHCC.2014.6883013

53. Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. 2011. Bricolage: Example-based Retargeting for Web Design. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2197–2206. http://doi.org/10.1145/1978942.1979262

54. Barry R Lawson, Kenneth R Baker, Stephen G Powell, and Lynn Foster-Johnson. 2009. A comparison of spreadsheet users with different levels of experience. *Omega* 37, 3: 579–590. http://doi.org/http://dx.doi.org/10.1016/j.omega.2007.12.004

55. Avraham Leff and James T Rayfield. 2001. Web-Application Development Using the Model/View/Controller Design Pattern. *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, IEEE Computer Society, 118–127. Retrieved from http://dl.acm.org/citation.cfm?id=645344.650161

56. Mark Levene and George Loizou. 1994. 30th IEEE Conference on Foundations of Computer Science The nested universal relation data model. *Journal of Computer and System Sciences* 49, 3: 683–717. http://doi.org/http://dx.doi.org/10.1016/S0022-0000(05)80076-5

57. Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-User Development: An Emerging Paradigm. In *End User Development SE - 1*, Henry Lieberman, Fabio Paternò and Volker Wulf (eds.). Springer Netherlands, 1–8. http://doi.org/10.1007/1-4020-5386-X_1

58. James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. 2009. End-user programming of mashups with vegemite. *Proceedings of the 14th international conference on Intelligent user interfaces*, ACM, 97–106. http://doi.org/10.1145/1502650.1502667

59. Alex MacCaw. 2011. *JavaScript Web Applications*. " O'Reilly Media, Inc."

60. C Mohan. 2013. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. *Proceedings of the 16th International Conference on Extending Database Technology*, ACM, 11–16. http://doi.org/10.1145/2452376.2452378

61. Brad A Myers, Margaret M Burnett, Andrew J Ko, Mary Beth Rosson, Christopher Scaffidi, and Susan Wiedenbeck. 2010. End User Software

Engineering: CHI 2010 Special Interest Group Meeting. *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, ACM, 3189–3192. http://doi.org/10.1145/1753846.1753953

62. Brad A Myers, Dario A Giuse, Roger B Dannenberg, et al. 1990. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer* 23, 11: 71–85. http://doi.org/10.1109/2.60882

63. Brad A Myers, Richard G McDaniel, Robert C Miller, et al. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Trans. Softw. Eng.* 23, 6: 347–365. http://doi.org/10.1109/32.601073

64. Brad A Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 59–66. http://doi.org/10.1145/22627.22349

65. Brad A Myers. 1991. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 243–249. http://doi.org/10.1145/108844.108903

66. Brad Myers, Scott E Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1: 3–28. http://doi.org/10.1145/344949.344959

67. Bonnie A Nardi, James R Miller, and David J Wright. 1998. Collaborative, programmable intelligent agents. *Commun. ACM* 41, 3: 96–104. http://doi.org/10.1145/272287.272331

68. Bonnie A Nardi and James R Miller. 1990. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development. *Proceedings of the 1990 ACM Conference on Computer-supported Cooperative Work*, ACM, 197–208. http://doi.org/10.1145/99332.99355

69. Bonnie A Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA.

70. Stephen Oney, Brad Myers, and Joel Brandt. 2012. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ACM, 229–238. http://doi.org/10.1145/2380116.2380146

71. Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: A Language and Environment for Expressing Interface Behavior. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ACM, 263–272. http://doi.org/10.1145/2642918.2647358

72. Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. *Proceedings of the 2009 IEEE Symposium on*

Visual Languages and Human-Centric Computing (VL/HCC), IEEE Computer Society, 105–108. http://doi.org/10.1109/VLHCC.2009.5295287

73. Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. *Proceedings of the 27th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc., 381–390. Retrieved from http://dl.acm.org/citation.cfm?id=645927.672045

74. Jochen Rode, Yogita Bhardwaj, ManuelA. Pérez-Quiñones, MaryBeth Rosson, and Jonathan Howarth. 2005. As Easy as "Click": End-User Web Engineering. In *Web Engineering SE - 61*, David Lowe and Martin Gaedke (eds.). Springer Berlin Heidelberg, 478–488. http://doi.org/10.1007/11531371_61

75. Mary Beth Rosson, Julie Ballin, and Jochen Rode. 2005. Who, what, and how: a survey of informal and professional Web developers. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, 199–206. http://doi.org/10.1109/VLHCC.2005.73

76. Mike Samuel, Prateek Saxena, and Dawn Song. 2011. Context-sensitive Auto-sanitization in Web Templating Languages Using Type Qualifiers. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 587–600. http://doi.org/10.1145/2046707.2046775

77. Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, 207–214. http://doi.org/10.1109/VLHCC.2005.34

78. Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8: 57–69. http://doi.org/10.1109/MC.1983.1654471

79. Timothy Sohn, Kevin A Li, William G Griswold, and James D Hollan. 2008. A Diary Study of Mobile Information Needs. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 433–442. http://doi.org/10.1145/1357054.1357125

80. Tableau Software. 2014. Tableau. Retrieved from http://www.tableausoftware.com/

81. Steven L Tanimoto. 2013. A perspective on the evolution of live programming. *Live Programming (LIVE), 2013 1st International Workshop on*, 31–34. http://doi.org/10.1109/LIVE.2013.6617346

82. Rattapoom Tuchinda, Pedro Szekely, and Craig A Knoblock. 2008. Building Mashups by example. *Proceedings of the 13th international conference on Intelligent user interfaces*, ACM, 139–148. http://doi.org/10.1145/1378773.1378792

83. Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. 2014. Stream Processing with a Spreadsheet. In *ECOOP 2014 – Object-Oriented Programming SE - 15*, Richard Jones (ed.). Springer Berlin Heidelberg, 360–384. http://doi.org/10.1007/978-3-662-44202-9_15

84. Fernanda B Viegas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. 2007. ManyEyes: a Site for Visualization at Internet Scale. *Visualization and Computer Graphics, IEEE Transactions on* 13, 6: 1121–1128. http://doi.org/10.1109/TVCG.2007.70577

85. Amy Voida, Ellie Harmon, and Ban Al-Ani. 2011. Homebrew Databases: Complexities of Everyday Information Management in Nonprofit Organizations. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 915–924. http://doi.org/10.1145/1978942.1979078

86. Miriam Walker, Leila Takayama, and James A Landay. 2002. High-fidelity or low-fidelity, paper or computer? Choosing attributes when testing web prototypes. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 661–665.

87. Guiling Wang, Shaohua Yang, and Yanbo Han. 2009. Mashroom: End-user Mashup Programming Using Nested Tables. *Proceedings of the 18th International Conference on World Wide Web*, ACM, 861–870. http://doi.org/10.1145/1526709.1526825

88. Lutz Wegner, Sven Thelemann, Jens Thamm, Dagmar Wilke, and Stephan Wilke. 1997. Navigational Exploration and Declarative Queries in a Prototype for Visual Information Systems. In Clement Leung (ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–218. http://doi.org/10.1007/3-540-63636-6_12

89. Nicholas Wilde and Clayton Lewis. 1990. Spreadsheet-based Interactive Graphics: From Prototype to Tool. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 153–160. http://doi.org/10.1145/97243.97268

90. David Wolber, Yingfeng Su, and Yih Tsung Chiang. 2002. Designing Dynamic Web Pages and Persistence in the WYSIWYG Interface. *Proceedings of the 7th International Conference on Intelligent User Interfaces*, ACM, 228–229. http://doi.org/10.1145/502716.502770

91. Jeffrey Wong and Jason I Hong. 2007. Making mashups with marmite: towards end-user programming for the web. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 1435–1444. http://doi.org/10.1145/1240624.1240842

92. Jeffrey Wong and Jason Hong. 2008. What do we "mashup" when we make

mashups? *Proceedings of the 4th international workshop on End-user software engineering*, ACM, 35–39. http://doi.org/10.1145/1370847.1370855

93. Alec Woo, Siddharth Seth, Tim Olson, Jie Liu, and Feng Zhao. 2006. A spreadsheet approach to programming and managing sensor networks. *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, 424–431. http://doi.org/10.1109/IPSN.2006.243910

94. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 423–438. http://doi.org/10.1145/2517349.2522737

95. Nan Zang and M B Rosson. 2008. What’s in a mashup? And why? Studying the perceptions of web-active end users. *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, 31–38. http://doi.org/10.1109/VLHCC.2008.4639055

96. Nan Zang, Mary Beth Rosson, and Vincent Nasser. 2008. Mashups: who? what? why? *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, ACM, 3171–3176. http://doi.org/10.1145/1358628.1358826

97. Kibana. Retrieved from https://www.elastic.co/products/kibana

98. StreamBase. Retrieved from http://www.streambase.com/

99. Introduction to queries - Access - Office Support. Retrieved from https://support.office.com/en-us/article/Introduction-to-queries-d85e4893-0ed7-4118-8297-785a01357516

100. SQLyog. Retrieved from https://www.webyog.com/#Slides_SQLyog

101. SQLeo Visual Query Builder. Retrieved from https://sourceforge.net/projects/sqleo/

102. Yahoo Pipes. Retrieved from http://pipes.yahoo.com/

103. Introduction to Microsoft Power Query for Excel. Retrieved from https://support.office.com/en-us/article/Introduction-to-Microsoft-Power-Query-for-Excel-6e92e2f4-2079-4e1f-bad5-89f6269cd605

104. AngularJS. Retrieved from https://angularjs.org/

105. Handlebar.js. Retrieved from http://handlebarsjs.com/

106. Mustache.js. Retrieved from https://github.com/janl/mustache.js/

107. Underscore.js. Retrieved from http://underscorejs.org/

108. React. Retrieved from https://facebook.github.io/react/

109. JSON.simple. Retrieved from https://github.com/fangyidong/json-simple

110. GSON. Retrieved from https://github.com/google/gson
111. FasterXML/Jackson. Retrieved from https://github.com/FasterXML/jackson
112. Jansson. Retrieved from http://www.digip.org/jansson/
113. jsonQ. Retrieved from https://github.com/s-yadav/jsonQ
114. View and Analyze Live Streaming Data in Excel. Retrieved from https://developer.ibm.com/bluemix/2015/08/10/view-analyze-live-streaming-data-excel/
115. Use Stream Analytics to feed Power BI from Application Insights. Retrieved from https://azure.microsoft.com/en-us/documentation/articles/app-insights-export-power-bi/
116. PipelineDB. Retrieved from https://www.pipelinedb.com/
117. Active Query Builder. Retrieved from http://www.activequerybuilder.com/
118. Scraper. Retrieved from http://mnmldave.github.io/scraper/
119. ScraperWiki. Retrieved from https://scraperwiki.com/

APPENDIX A. A LIST OF SUPPORTED SPREADSHEET FUNCTIONS IN GNEISS

As described in Section 5.5, for the user study that evaluates Gneiss's features for using hierarchical data, I gave Gneiss participants a list of spreadsheet functions that they could use in the study. The list I gave the participants has the first 12 functions in the "Conventional spreadsheet function" table (from `SUM` to `LOOKUP`), and the first function (`IF`) in the "Extended conventional spreadsheet function" table. I did not show participants the other functions as they are for using streaming data or programming interactive behaviors in web applications and are clearly not relevant to the study tasks.

Conventional spreadsheet function	
`SUM(`*`number1, [number2], …`*`)`	The function returns the sum of the numbers
`AVERAGE(`*`number1, [number2], …`*`)`	The function returns the average of the numbers
`MAX(number1, [number2], …)`	The function returns the largest number of the numbers
`CONCATENATE(string1, [string2], …)`	The function returns the concatenated string of all input strings.
`ISBLANK(cell)`	The function returns true if `cell` is blank, false otherwise.
`AND(condition1, [condition2], …)`	The function returns true if all conditions are true, and false otherwise
`OR(condition1, [condition2], …)`	The function returns true if any of the conditions is true, and false otherwise
`INDEX(array, index)`	The function returns the value in `array` at index `index`.
`COUNT(value1, [value2], …)`	The function returns the number of values
`COUNTUNIQUE(value1, [value2],…)`	The function returns the number of unique values
`COUNTIF(range, criteria)`	The function returns the number of cells in `range` that match `criteria`.
`LOOKUP(lookup_value, lookup_array, result_array)`	The function finds the first index of the value `lookup_value` appears in `lookup_array`, and returns the item in that index in the `result_array`. If the function can't find `lookup_value` in `lookup_array`, it returns an empty string.
`DAY(time_value)`	The function returns the day number (1-31)

	of `time_value`
`HOUR(time_value)`	The function returns the hour number (0-23) of `time_value`
`IMAGE(url)`	The function displays the image in `url` in the spreadsheet cell
Extended conventional spreadsheet functions *(with the extended features in italic)*	
`IF(condition, value1, [value2])`	If `condition` is true, the function returns `value1`. Otherwise, the function returns `value2` if `value2` is specified, *or the cell's current value if `value2` is omitted.*
`NOW()`	The function returns the current date and time *and updates every minute*.
`TODAY()`	The function returns the current data *and updates daily.*
New Gneiss spreadsheet functions	
`TIMER(exp, ms)`	The function returns the result of `exp` and updates every `ms` milliseconds.
`ANIMATE(start_value, end_value, ms)`	The function's return value gradually increases from `start_value` to `end_value` in `ms` milliseconds. It uses jQuery's animation algorithm to animate the value.
`REFRESH(exp)`	The function returns the value of `exp` and invalidates all dependent cells/constraints, forcing them to be recomputed.
`FETCHTIME(cell)`	If `cell` has data retrieved from a web service, the function returns the data's retrieved time. Otherwise, the function returns a blank string.
`SELECTBYTIME(range, start_time, end_time)`	The function returns an array of values that are in `range` and retrieved between `start_time` and `end_time`.

APPENDIX B. A LIST OF SUPPORTED WEB GUI ELEMENTS IN GNEISS

Interactive properties whose values are not editable but change based on how users interact with the web application are shown in *blue color and italic*. As described in Chapter 4, most GUI elements have a *State* interactive property that shows how the mouse cursor interacts with the element. Possible values for *State* are "idle", "hover", "pressed", and "clicked".

Element Name	Properties
Text and Image	
Text	Value, Color, FontSize, FontStyle, FontWeight, TextDecoration, Link (page name or URL to make the element become a hyperlink), *State*
Heading	Value, Color, FontSize, Link, *State*
Image	Source (URL that is the source of the image), AltText, Width, Height, Border, Link, *State*
Input	
Text box	*Value*, PlaceHolder (grey text shown in the textbox when the textbox is empty), Width, Height, Live (if "true", *Value* changes every time the user presses a key. If "false", *Value* changes when the user presses the enter key. Default is "false"), *State*
Slider	*Value*, Max, Min, Step, Width, Live (if "true", *Value* changes as the user drags the slider. If "false", *Value* changes when the user releases the mouse. Default is "false"), *State*
Radio Button	*Checked* ("true" or "false"), Group (the radio button group the button is in. When multiple radio buttons have the same Group value, checking one button will uncheck all other buttons. Default is "Group1".), Label (radio button text), *State*
Checkbox	*Checked* ("true" or "false"), Label (checkbox text), *State*
Button	Width, Height, Value (button text), Link, *State*
List	
Vertical List	BulletStyle, Populate, NumberOfItems
Grid List	ItemWidth, ItemHeight, Populate, NumberOfItems
Visualization (Implemented using Google Visualization API)	
Map	Addresses, Latitudes, Longitudes, TooltipText, Width, Height, Border, *State*
Line Chart	Title, Data, AxisLabels, HAxisTitle, VAxisTitle, Width, Height, Border, *State*
Bar Chart	Title, Data, AxisLabels, HAxisTitle, VAxisTitle, TooltipText, Width, Height, Border, *State*

Scatter Chart	Title, DataX, DataY, HAxisTitle, VAxisTitle, TooltipText, Width, Height, Border, *State*
Treemap	Title, Data, Width, Height, Border, *State*

APPENDIX C. POST-STUDY SURVEY FOR THE USER STUDY

1. Demographic information

Your age: _____

Your gender (circle): Female Male

Your highest degree so far (circle):
PhD Master BS/BA High school Other_____

Please rate your experience with programming in general

1	2	3	4	5
No experience				Very experienced

What is the programming language that you are most familiar with? _____

What is the editor you usually use to write code?
_____

Please rate your proficiency with Microsoft Excel

1 None
2 Novice – know how to enter, sort, filter and visualize data
3 Intermediate – know how to use basic spreadsheet functions such as SUM, AVERAGE, IF.
4 Advanced – know how to use pivot tables and advanced spreadsheet functions such as LOOKUP
5 Superior – know how to write macros using VBA

Please rate your experience with using SQL

1	2	3	4	5
No experience				Very experienced

Please rate your experience with using JavaScript

1	2	3	4	5
No experience				Very experienced

Please rate your experience with using Python

1	2	3	4	5
No experience				Very experienced

Please rate your experience with using JSON data

1	2	3	4	5
No experience				Very experienced

Are there other tools/languages besides what you used for the study that you often use to analyze data? Please list them here.

_____

2. Please rate what you used for the study

Ease of learning it
(If you already knew how to use it before the study, try to think back to when you were learning it.)

1	2	3	4	5	6	7
Very hard						Very easy

Ease of using it

1	2	3	4	5	6	7
Very hard						Very easy

Usefulness in helping you complete the tasks

1	2	3	4	5	6	7
Not useful at all						Very useful

3. Please rate the tasks you did

Do the tasks seem realistic to you?

1	2	3	4	5	6	7
Not realistic at all						Very realistic

In real life, how often do you do data exploration tasks similar to today's tasks?

1	2	3	4	5	6	7
Never did anything similar						Often did similar tasks

Difficulty of the tasks using the given application/language

1	2	3	4	5	6	7
Not difficult at all						Very difficult

4. Can you imagine using a tool like this in your own work? *(Only the Gneiss group saw this question)*

1	2	3	4	5	6	7
No, not at all						Yes, I would use it frequently

5. The tool you used is a research prototype. What are some features or changes you would like to see? *(Only the Gneiss group saw this question)*

_____

_____

_____

_____